Resource Signal Prediction and Its Application to Real-time Scheduling Advisors

Peter August Dinda

May, 2000

CMU-CS-00-131

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Thesis Committee:
David O'Hallaron, Chair
Thomas Gross
Peter Steenkiste
Jaspal Sublok, University of Houston
David Bakken, Washington State University

20000926 029

DTIC QUALITY INSPECTED 4

Carnegie Mellon

School of Computer Science

DOCTORAL THESIS
in the field of
COMPUTER SCIENCE

Resource Signal Prediction and Its Application To Real-time Scheduling Advisors

PETER A. DINDA

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

_____   2/14/2000
THESIS COMMITTEE CHAIR   DATE

_____   5/18/00
DEPARTMENT HEAD   DATE

APPROVED:

_____   5/18/00
DEAN   DATE

Abstract

A distributed interactive application spawns resilient real-time tasks with known resource requirements in response to aperiodic user actions. When running in a shared computing environment that supports neither reservations nor globally-respected priorities, such an application must carefully choose which host runs a task in order to increase the chances that the task's deadline will be met.

A real-time scheduling advisor is a middleware service that the application can use to find the most appropriate host for the task. In addition to recommending a host, the advisor also predicts the running time of the task on that host. The application uses this feedback to modify the task's resource requirements or deadline until a host is found where the task will meet its deadline with sufficiently high probability.

This dissertation recommends basing real-time scheduling advisors on the explicit prediction of resource signals, which are easily measured, time-varying, scalar quantities that are strongly correlated with resource availability. This resource-oriented approach has numerous advantages over the competing application-oriented approach, which I also studied. It scales well, makes decisions based on up-to-date information, can support other forms of adaptation advisors, and can easily leverage advances in statistical signal prediction techniques. However, resource signal predictions exist at considerable remove from predictions of application performance.

To show that this gap can be spanned, this dissertation describes the design, implementation, and performance evaluation of a prototype real-time scheduling advisor that is based on the prediction of host load signals. I have found that, despite its complex properties, which include self-similarity and epochal behavior, host load can be usefully predicted using linear time series models. These models have sufficiently low overhead to be used in practice, and I have developed a toolkit to make it easy to do so. Furthermore, I have devised an algorithm that uses host load predictions to compute a confidence interval for the running time of a task on a particular host. My real-time scheduling advisor uses these confidence intervals to provide useful recommendations to applications. Each layer of this online system has been evaluated using real host load signals and workloads.

Being able to predict the running time of a task is vital to controlling many different adaptation mechanisms in pursuit of goals other than simply those of the real-time scheduling advisor. For this reason, I also expose the running time advisor, the part of my system that computes a confidence interval for the running time of a task.

Acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Users demand responsiveness from interactive applications such as scientific visualization tools, image editors, modeling tools based on physical simulations, and games. Such applications react to aperiodically arriving messages that arise from user actions. In response to each message, the application program executes a task whose computation creates visual and aural feedback for the user. This feedback helps determine the subsequent actions of the user, and thus subsequent tasks. The aperiodicity in message arrival is a result of having a "human in the control loop." To be responsive, the application must execute the task induced by each message as quickly as the user reasonably expects. This timeliness requirement can be easily expressed as a deadline for the task. We assume that the application is resilient in the face of missed deadlines.

At one point, creating applications that were responsive in this sense was relatively easy since the user's machine was very predictable from the point of view of the programmer. Today, however, the user's machine is becoming increasingly less predictable due to operating system jobs, daemons, other users' jobs, and the like. These external factors make the actual computation rate the programmer can expect vary in complex ways. Further, the user's machine may simply not be fast enough or have enough memory to perform some computations responsively.

On the other hand, the user's machine is no longer alone—there are many other hosts on the local area network which it can talk to. As the overheads of remote execution facilities such as remote procedure call (RPC) systems, object request brokers (ORBs), and distributed shared memory (DSM) systems decline, it becomes more appealing to execute tasks remotely to achieve the responsiveness that interactive applications require. Interactive applications are thus becoming distributed interactive applications. Unfortunately, the overwhelming majority of networks and hosts do not provide any sort of resource reservations, or even priority-based scheduling that a programmer could build upon to make a group of hosts behave in a more predictable fashion than an individual host. However, these shared, unreserved environments can be measured with increasingly sophisticated tools.

The ability to run a task on any host in the environment greatly increases the *opportunity* for the task to meet its deadline. However, to exploit this opportunity, the application must *choose* an appropriate host, which can be difficult. A real-time scheduling advisor is a middleware service that advises the application as to the host where the task's deadline is most likely to be met. It may also provide additional information, such as the predicted running time of the task on the proffered host, which the application can use to adapt in other ways.

A real-time scheduling advisor bases its advice on resource measurements, the application's characterization of the task's resource demands, and the required deadline. Because reservations are unavailable in the computing environment, this advice comes with no guarantees—a real-time scheduling advisor operates on a best-effort basis. The usefulness of this service then depends on its measured performance. This dissertation shows that the measured performance of a real-time scheduling advisor running in real computing

1

environments can be quite impressive. The real-time scheduling advisors we developed can greatly increase the probability that a task's deadline is met. Furthermore, they can accurately predict the performance of tasks *before* they are run, thus giving the application a chance to adapt in different ways when insufficient resources are available to meet the original deadline. Finally, they can introduce appropriate randomness into their scheduling decisions, thus allowing advisors to operate obliviously of each other with a low probability of disastrous interaction due to unforeseen feedback loops.

The design space for real-time scheduling advisors is vast, but most designs involve predicting the performance, either explicitly or implicitly, of the task on each of the prospective hosts. The performance is determined predominantly by resource availability. Designs that use explicit prediction are sub-divided into resource-oriented prediction approaches and application-oriented prediction approaches. Resource-oriented approaches predict future resource availability using information available about the resource. These predictions of resource availability, and the task's resource demands are then supplied to a model that estimates the task's performance. Application-oriented approaches predict task performance directly using application information such as the performance of previous tasks.

This dissertation argues for basing real-time scheduling advisors on explicit resource-oriented prediction, specifically on the prediction of resource signals. This approach has much to recommend it over the application-oriented prediction approach:

- It scales better.

- Multiple applications or nodes of a single application can easily share the same predictions.

- It operates independently of application execution and thus can always provide the latest information about any resource.

- It can provide the basis for other kinds of scheduling advisors and quality of service predictors.

- It can more easily leverage advances in the statistical signal processing and time series analysis communities.

In contrast to the application-oriented approach, the resource-oriented approach predicts quantities that are at some remove from those that concern the application. This dissertation demonstrates that it is possible to span this gap between resource predictions and task performance. This enables effective real-time scheduling advisors based on explicit resource-oriented prediction.

The core of the dissertation describes the design, implementation, and evaluation of a real-time scheduling advisor for compute-bound tasks. The advisor is based on explicit resource-oriented prediction using the techniques of linear time series analysis to predict available CPU time. The resource signal is host load (specifically, the Digital Unix five second load average), which we found to correlate very well with available CPU time.

Our explicit resource-oriented prediction approach is based on statistical signal processing of resource signals. Resource signals are sequences of periodic measurements that are strongly correlated with the availability of some underlying resource. We exploit linear time series analysis (for the most part) to characterize resource signals and find appropriate predictors for them. We developed a methodology for the process and a toolkit that facilitates carrying out the methodology and implementing on-line resource prediction systems for new resource signals.

The main contribution of this dissertation is to show that the resource-oriented prediction approach can work—that the application of the resource signal methodology to host load results in useful predictions that can be projected up to the application in a manner that is sufficient to drive adaptation decisions. The projection takes the form of a query interface through which an application can request a qualified prediction

(in the form of a confidence interval) for the running time of a task. This fundamental information is useful for controlling many different adaptation mechanisms to pursue many different goals. We show that it is sufficient to control one particular mechanism (choice of host) to achieve one particular goal (meeting a deadline). Effectively, we show that real-time scheduling advisors based on explicit resource-oriented prediction using statistical signal processing are both feasible and powerful.

In the remainder of this chapter, we first describe the application domain for which real-time scheduling advisors are intended and provide examples of applications from this domain. Next, we describe the characteristics of the computing environment we target, and the scheduling problem induced by running our applications in such environments. After this, we outline the design space for real-time scheduling advisors, illustrating the advantages and disadvantages of both the resource-oriented and application-oriented approaches. The main observation is that the resource-oriented approach—specifically, an approach based on resource signal prediction—is preferable to the application-oriented approach provided that the gap from resource signal predictions to application-level predictions can be spanned. We then outline the prototype real-time scheduling advisor that forms the core of the dissertation, and describe the resource signal methodology we used to develop it, and which we shall apply to signals other than host load in the future. Finally, we outline the remaining chapters of the dissertation.

1.1 Applications

A real-time scheduling advisor operates on the behalf of a distributed interactive application. In such applications, computation takes the form of tasks that are initiated by aperiodic user actions. Each task produces feedback for the user which helps determine his next action. For this reason, the task must be completed in timely manner soon after it has been initiated. The application expresses this timeliness requirement in the form of a deadline for each task. The application is resilient in the face of a missed deadline. The real-time scheduling advisor suggests which of a set of hosts is most appropriate to run the task. In addition to this required form of adaptation, the application may also be able to adapt by changing the compute requirements of the task or the required deadline.

The remainder of this section describes the characteristics of the applications that this thesis targets and their execution model in more detail. In addition, we present four applications that conform to the characteristics and the model.

1.1.1 Characteristics

The applications we are interested in supporting have the following characteristics.

Interactivity

The application is interactive—computation takes the form of tasks that are initiated or guided by a human being who desires responsiveness. Achieving responsiveness amounts to providing timely, consistent, and predictable feedback to individual user actions. If the feedback arrives too late or there is too much jitter for a series of similar actions, the utility of the program is degraded, perhaps severely. Research has shown that people have difficulty using an interactive application that does not respond in this manner [41, 72]. Our mechanism for specifying timely, consistent, predictable feedback is the task deadline.

Aperiodicity

The application's tasks arise from aperiodic user actions. The aperiodicity is due to from the variable "think time" humans need to decide their next action [42]. Aperiodicity precludes such traditional real-time

approaches as rate monotonic algorithms [81].

Sequentiality

The application has only a single task outstanding at any time. The user needs the feedback produced by that task to determine his next action and its resulting task. We discuss ways of loosening this restriction in the concluding chapter.

Resilience

The application is resilient in the face of missed deadlines to the degree that it does not require either statistical or deterministic guarantees from the real-time system. The inability to meet a deadline does not make the application unusable, but merely results in lowered utility. For example, occasional missing frames in playing back video do not make the video performance unacceptable. Consistently missing or irregular frames, however, result in unacceptable playback. Resilience is the characteristic that enables the best-effort semantics of real-time scheduling advisors as opposed to traditional "soft" (statistically guaranteed) [136, 19, 68] semantics and "hard" (deterministically guaranteed) real-times semantics [128, 119].

Distributability

The application has been developed with distributed, possibly parallel, operation in mind. We assume that it is possible to execute its tasks on any of the available hosts using, for example, mechanisms such as CORBA [98] or Java RMI [129]. Tasks need not be replicable (stateless), but any data movement required to execute a task on a particular host must be exposed, for example, through CORBA's Object By Value mechanism [98, Chapter 5], or by Java's serialization and reflection mechanisms. The core of this dissertation concentrates on compute-bound tasks.

Adaptability

In addition to the adaptability provided by being able to choose on which of a set of hosts a task will execute, our target applications may also be able to indirectly adjust the amount of computation and communication resources a task requires. Adjustments such as changing resolution, image quality, frame rate, and deadline may be needed in order to deal with situations where a task's deadline cannot be met by choosing the appropriate host to run it, or when longer term changes in the available resources result in many tasks missing their deadlines.

Compute-bound tasks

In the core of this dissertation we restrict ourselves to compute-bound tasks. This is because the proof-of-concept system that we develop focuses only on CPU time. The restriction is not inherent to real-time scheduling advisors based on explicit resource-oriented prediction using statistical signal processing. Our RPS toolkit already includes facilities for network resource prediction based on Remos [82] network measurements, and in the concluding chapter we present our current work in evaluating the prospects for network prediction.

1.1.2 Execution model

Our model interactive application has a very simple main loop that waits for aperiodically arriving user input and then issues an appropriate task. The task runs to completion, producing feedback to the user. After the

Symbol	Explanation
t_{now}	Arrival time of the task
t_{nom}	Compute requirements of the task (nominal running time)
$slack$	Permitted expansion factor of the task
$t_{now} + (1 + slack)t_{nom}$	Deadline of the task
t_{act}	The actual running time of the task
$t_{now} + t_{act}$	The actual completion time of the task

Table 1.1: Elements of the task execution model.

task completes and the feedback is delivered, the user may produce other input, resulting in a new task being initiated. Table 1.1 summarizes the symbols we use in this section to describe the execution model.

From the perspective of the real-time scheduling advisor, a task arrives at the current time, t_{now}. The application specifies the compute requirements of the task in terms of its nominal running time, t_{nom}. The nominal running time is the time the task would take to run on a host with no other extant work. We assume that the task is compute-bound and that the communication time required to start the task running on a particular host is negligible compared to the nominal time of the task. In the concluding chapter, we consider how to extend the resource-oriented approach to include communication costs where this is not the case. The application expresses the deadline of the task in the form of the $slack$, or the maximum additional expansion of the running time of the task. Because the task must be immediately executed, the deadline is then $t_{now} + (1 + slack)t_{nom}$. The actual running time of the task, is t_{act}, so its completion time is $t_{now} + t_{act}$. Thus, the deadline is met if $t_{act} \leq (1 + slack)t_{nom}$.

1.1.3 Examples

In this section, we introduce four examples of applications which exhibit the characteristics we described earlier and which could be executed according the execution model. The examples include QuakeViz, a scientific visualization tool, OpenMap, a geographic information services tool, an acoustic modeling tool, and an image editor. Of these, we consider QuakeViz in the most detail, including measurements of the compute requirements of two QuakeViz applications and a description of the active frame execution environment which we have helped to develop for such applications. OpenMap is representative of applications which use replicated servers. Other examples include the mirroring and anycast of web content [93], and distributed database systems [53, 56]. The acoustic modeling application is a computer aided design application based on a physical simulation and can also be seen as a computational steering application. Image editing is typical of the large document editing applications common on personal computers.

The purpose of this section is to illustrate applications in which a real-time scheduling advisor can provide benefits. The focus of the thesis, however, is not on any particular application, but rather using an explicit resource-oriented prediction-based approach to solving the general scheduling problem posed by real-time scheduling advisors. Our evaluation does not focus on any particular application, but is based on randomized tasks whose nominal compute times are chosen to be within the range of interest for the QuakeViz tasks we describe here.

QuakeViz

The Quake project developed tools to perform detailed simulations of large geographic areas during strong earthquakes [9] and then applied those tools to simulate earthquakes in various earthquake prone areas. These simulations produce vast amounts of output data. For example, to simulate the response of the San

(a) Volume visualization   (b) Isosurface visualization
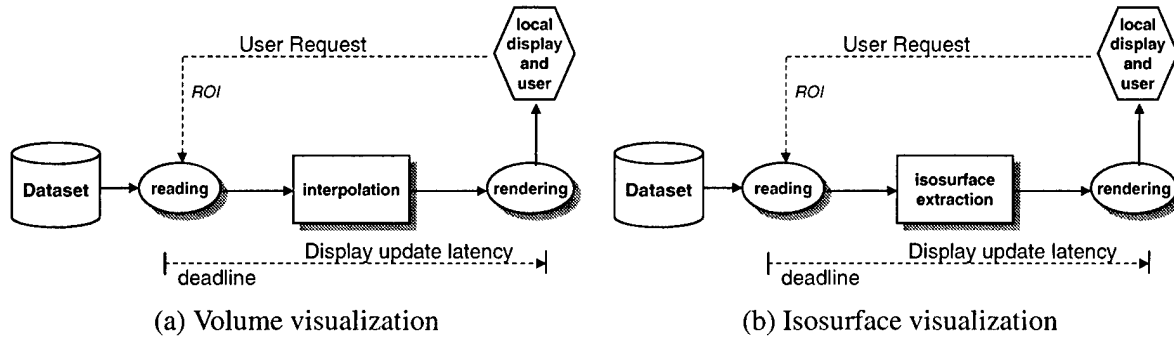
Figure 1.1: Example QuakeViz applications: (a) Volume visualization using a structured grid. (b) Isosurface visualization using an unstructured grid.

Fernando Valley to an aftershock of the 1994 Northridge Earthquake required a data representation containing 77 million tetrahedrons, produced over 40 million unknowns per time step, and resulted in over 6 TB of output data.

Interactive scientific visualization [58] is a necessary prerequisite in order for humans to make use of these colossal datasets. Unfortunately, current visualization systems for such datasets either require extremely expensive hardware, or are batch-oriented. To remedy this situation, the Dv group, a part of the Quake project which includes this author, is designing and building a framework for constructing interactive scientific visualizations of large datasets that can run on shared, unreserved distributed computing environments [3]. Visualizations of datasets produced by the Quake Project, or QuakeViz applications, are the first target of the Dv framework.

The Dv framework is based on the active frame model. Active frames are a form of active messages [135] in that they contain both data and a program for transforming that data. A QuakeViz application can be expressed as a flowgraph whose nodes represent computationally expensive data transformations and whose edges represent communication. The flowgraph source is the server which provides the Quake dataset and its sink is the user's workstation. Typically, the flowgraph is linear. The user initiates computation by sending an active frame to the server. The active frame contains the flowgraph of the computation the user requires, a specification of the region of the dataset the user is interested in, and a deadline for when the result must be displayed. The first node of the flowgraph executes on the server and copies the necessary data into the active frame. The frame then sends itself to the most appropriate host to execute the next node in the flowgraph. The frame uses the real-time scheduling advisor to decide which host is the most appropriate. This continues until the last node of the flowgraph has been executed and the result has been displayed on the user's workstation.

Figure 1.1 shows the flowgraphs of two simple QuakeViz applications. More complex flowgraphs are discussed elsewhere [3, 29]. In both applications, the dataset produced by the simulation contains values whose coordinates are based on irregular mesh. A QuakeViz application can maintain this unstructured representation throughout its flowgraph, or it can interpolate the data onto a regular grid in order to make down-stream processing faster and reduce the volume of communication. Figure 1.1(a) shows a simple volume visualization that shows the data in the region of interest as a 3-dimensional image. The data is interpolated to a regular grid in order to make the rendering computation faster. Figure 1.1(b) shows a visualization in which isosurfaces corresponding to various response intensities in the region of interest are displayed. In this case, the data is left in its unstructured form.

Figure 1.2 shows the compute requirements for the flowgraph shown in Figure 1.1(a) as a function of the size of the structured grid. The computation here is measured as the user time for a sequential version of the visualization written in vtk [118] and running on a 500 MHz Alpha 21164 machine under Digital Unix
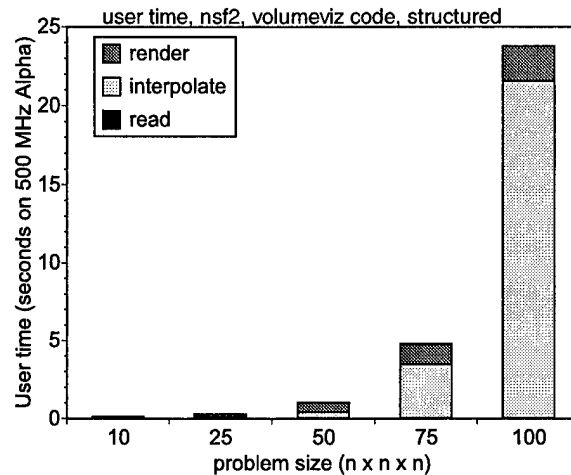
Figure 1.2: Compute requirements of the volume visualization code of Figure 1.1(a).

| (a) isosurface extraction | (b) rendering |
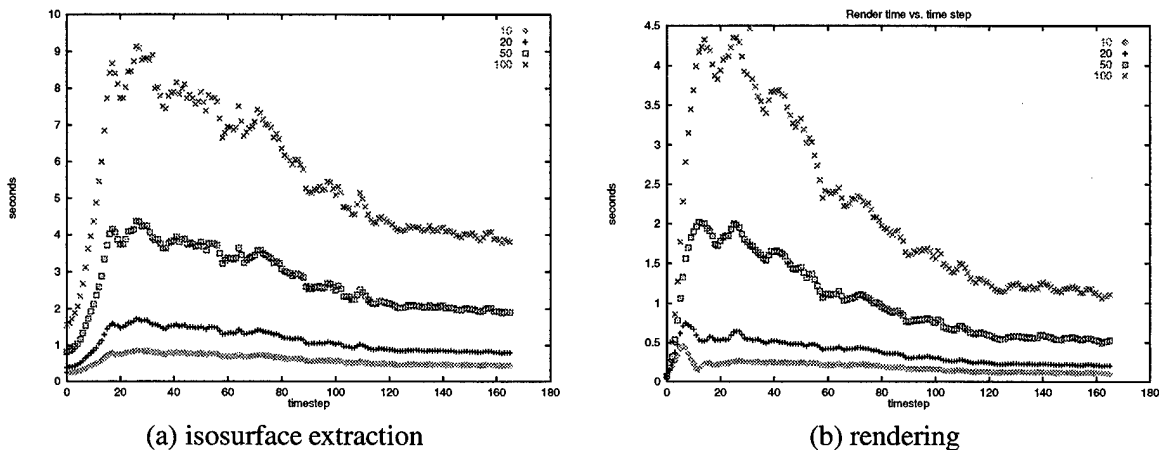
Figure 1.3: Compute requirements versus timestep of the isosurface visualization code of Figure 1.1(b).

4.0D. As we can see, the computation involved at each stage of the flowgraph is significant and is usually dominated by the interpolation step, which the active frame model permits us to locate on any host. The read and interpolation steps must be sited at the server machine and the user's machine, respectively.

The amount of computation performed by the isosurface visualization code depends not only on the spatial region of interest, but also on the time step. Figure 1.3 shows how the compute requirements of the (a) isosurface extraction and (b) rendering steps of the isosurface visualization code in Figure 1.1(b) vary with the time step for different problem sizes. The measured code is a sequential version written in vtk and running on a 200 MHz Pentium Pro machine under Microsoft Windows NT 4.0. Measurements are of the user time. Again, we can see a computation (isosurface extraction) which requires significant CPU time and which can potentially be run on any host in the environment. It is also interesting to note that the compute requirements are quite predictable from time step to time step.

OpenMap

BBN's OpenMap is a architecture for combining geographical information from a variety of different, separately developed sources in order to present a unified coherent visual representation, in the form of a multi-
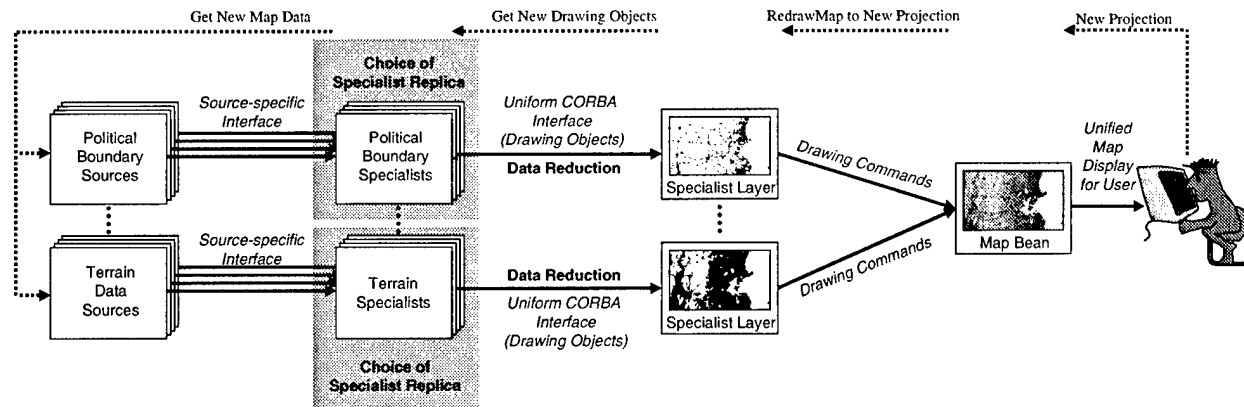
Figure 1.4: Structure of an OpenMap application. Solid arrows represent the flow of data, while dotted arrows represent user requests.

layered map, to the end user [13, 70]. OpemMap-based applications have been used to help coordinate U.S. military actions in the former Yugoslavia.

OpenMap consists of four different kinds of components. Geographical information is provided by third party *data sources*, which have unique interfaces. A *specialist* encapsulates a specific data source, hiding the details of accessing it behind a uniform CORBA interface. The interface is based on sequences of objects to be drawn. A specialist has a corresponding *layer* that draws an individual map based on the drawing objects. Finally, a *map bean* manages a group of layers, overlaying their maps to produce a single combined map for the user. Map beans and layers are Java Beans, which can be conveniently embedded into Java applications.

In Figure 1.4, we show the structure of an example OpenMap application where information from separate terrain and political boundary data sources are combined to present the user with a map of the Boston area. While the structure shown in Figure 1.4 appears at first glance to be a pipeline, it is important to note that it actually operates in a request-response manner. Computation happens only when the user decides to change the *projection* of the map (the set of layers and the region of the planet that is being viewed).

OpenMap is thus interactive—computation happens as a direct result of a projection change. To provide a good user experience, the time from a projection change to the resulting map display should be short, consistent, and predictable. A good abstraction for this requirement is a deadline placed on the computation initiated by a projection change. Achieving such deadlines is challenging because specialists and data sources may be located at distant sites and run on shared, unreserved hosts communicating via the Internet. However, missing OpenMap deadlines only degrades the user's experience—OpenMap is resilient.

The components of OpenMap were designed from the start to be physically distributed using CORBA communication mechanisms. We can use this enabler to build replicated specialists and data sources, as we highlight in gray in Figure 1.4. This provides a choice of which specialist is used to satisfy a projection change for a given layer. The real-time scheduling advisor can be used to decide which replica should be used. This functionality can even be hidden from the application by incorporating it into an object quality of service framework such as BBN's QuO [145]. In fact, as a proof of concept, we incorporated the host load prediction system described in this thesis into QuO as a system condition object and then developed QuO contracts that effectively represent a real-time scheduling advisor. This was then used to select the appropriate replica of an image server.

Acoustic CAD

Acoustic CAD involves designing a space (a room, or a loudspeaker enclosure, for example) using a CAD tool and being able to hear what such that space will sound like from different positions within it. For a given room configuration (room geometry and material properties, listener positions, and loudspeaker positions), we compute an impulse response function for each listener/loudspeaker pair. For a particular listener/loudspeaker pair, we convolve the music signal coming from the loudspeaker with the impulse response function, giving the room-filtered signal the listener would hear from that loudspeaker. By doing this for each loudspeaker and summing the resulting room-filtered signals, we simulate what the listener would hear given the room configuration [1].

The impulse responses are computed by simulating the wave equation using a finite difference method [125]. In steady state, periodic convolution and summation occurs to compute the sound output. When the user changes the room configuration by moving a loudspeaker, wall, or himself, the (expensive) computation of the impulse responses is repeated. It is these expensive user-initiated physical simulations that form the tasks for the real-time scheduling advisor.

As an alternative to sound, the user can view the impulse response functions directly, or can view the frequency response characteristics of the room computed from them. The user repeatedly adjusts the model parameters (furniture position and composition), simulates the physical system, and views the results. The goal is find a set of parameters that result in a flat response.

Image editor

Image editing gives the user tools to manipulate an in-memory visual image as a whole and in part. Some tools involve image-processing operations such as boxcar convolution on large regions of the image, while others involve emulating real-world tools such as pens, brushes, or spray paint cans. It is important to point out that image sizes are rapidly growing, and the limits imposed by photographic film, drum scanners, and digital cameras imply that truly vast (hundreds of megabytes to gigabytes) images will need to be edited. At the same time, the functionality of image editing software, in terms of the sophistication of image filters and how they can be applied is rapidly advancing.

The typical resolution of color reversal film is 100 lines per millimeter, which corresponds to 200 pixels per millimeter. With this information density, a 35mm slide contains 34.6 million pixels, or about 138 megabytes of information at 32 bits per pixel. A medium format 6 cm by 7 cm slide contains 168 million pixels or 672 megabytes, and the smallest large format slide, 4 inches by 5 inches, contains 516 million pixels or two gigabytes. These vast image sizes mean that even simple transformations result in large amounts of computation. However, the resolution at the user's workstation is limited by the screen resolution, which is much lower. This results in large computational requirements combined with potentially low communication requirements, which encourages a distributed implementation of image editing.

1.2 Shared, unreserved computing environment

Our machine model corresponds to the real world computing environments that most people have access to. In particular, we assume host computers interconnected by a local area network. The host computers have no centralized scheduler or coordinated scheduling mechanism and their local schedulers do not support reservations or real-time scheduling. Similarly, the network does not support any sort of reservation scheme. The hosts execute independent tasks that generate traffic on the network. This is, of course, a description of

---

[1]Ignoring the listener's Head Related Transfer Function, Doppler effects during movement, and other issues that are beyond the scope of this document. Interested parties should look at [14].

any modern group of workstations or PCs. The specific environments we studied are Digital Alpha-based workstations running Digital Unix 3.2 and 4.0. The software we developed has been ported to a variety of other Unix systems and Microsoft Windows NT.

In addition to these commonplace features, we also assume that it is possible to measure the system. In particular, it must be possible to acquire all the necessary permissions to record resource signals. In the case of the host load signal we use in this work, the baseline permission required is to run the Unix uptime command. On the Digital Unix machines we use, the permissions of a typical user permit the use of a much faster system call to measure the load, however. It is also necessary to have access to a real-time clock, through the Unix gettimeofday call, for example.

1.3 Scheduling problem

The scheduling problem that a real-time scheduling advisor attempts to solve is stated as follows. The real-time scheduling advisor operates on the behalf of a single application. The application needs to run tasks in response to aperiodically arriving user input. A task must run to completion before another task arrives, and the application can run the task on any of a set of shared, unreserved hosts. Suppose that a task arrives at the current time, t_{now}, and its compute requirement, expressed as a nominal running time on a quiescent host, is t_{nom}. The application wants the task to finish before the deadline $t_{now} + (1 + slack)t_{nom}$, where the *slack* is the maximum expansion factor that the application can allow. The problem the real-time scheduling advisor must solve is to choose the host from among the set of available hosts where the deadline is most likely to be met.

Ideally, the advisor will also inform the application whether it believes the deadline can be met on the chosen host. It may be the case that there are insufficient resources available on any host to meet the deadline. If this is the case, a prediction of the task's running time or whether it will meet its deadline or not enables the application to try a different form of adaptation or to change the deadline. One of the powerful aspects of the resource-oriented prediction approach described here is that it can provide this additional feedback.

It is also important to note that the predictions of running time that underly this approach are useful in achieving goals other than meeting deadlines. This is also an important feature of the resource-oriented approach.

1.4 Design space

The design space for real-time scheduling advisors that address the scheduling problem posed in Section 1.3 is vast. However, one characteristic that most designs necessarily share is the use of prediction, in that the advisor picks a host for the task based on some prediction of what the task's performance *will be* on each of the prospective hosts. This prediction can either be implicit or explicit. Explicit approaches attempt to directly predict some task performance metric—the task's running time, for example—for each of the hosts and then choose a host whose predicted performance is appropriate. In contrast, implicit approaches simply assume that the task's performance on each of the prospective hosts will be ordered according to some task-independent metric on the hosts, and then choose a host from early in the ordering.

Explicit prediction approaches are generally preferable to implicit approaches for a number of reasons. For one, they can provide additional value to the application. In particular, some explicit approaches, such as the one that forms the core of this dissertation, can inform the application as to whether the deadline is likely to be met, which provides the application with a chance to modify the task's requirements. Another advantage of explicit prediction is that it makes it possible to apply the prediction technology developed in

the statistics, signal processing, and artificial intelligence communities to the problem. The drawback is the explicit prediction approach is potentially much more complex than the implicit prediction approach.

Within the explicit prediction approach, there remain a vast number of design choices: What quantity will be predicted? What measurements will be used as the basis of the predictions? What prediction algorithm will be used? When will predictions happen? When will measurements be made? How will scheduling decisions be made using the predictions?

Although there are many possible answers to these questions, the primary distinction among explicit approaches is whether they are application-oriented or resource-oriented. In the application-oriented approach, the application measures the performance of each task it runs and provides this performance data to the advisor. The advisor predicts the performance of the next task on each of the hosts based on this history and then chooses an appropriate host based on the predictions. In the resource-oriented approach, each resource (eg, host) is conceptually responsible for measuring and predicting its own availability. When asked to schedule a task, the advisor collects the latest predictions of resource availability, uses them to compute predictions of some task performance metric (eg, running time) for each host, and then chooses the appropriate host based on those performance predictions.

The explicit application-oriented and resource-oriented prediction approaches are complementary in their potential advantages and disadvantages. The application-oriented approach has the advantage of operating directly on the performance metrics (whether deadlines are met, running time) that the application (and advisor) ultimately care about. However, the measurements and predictions made using this approach are entangled with application specifics and so are not useful to other applications. This leads to a duplication of effort, where each application is predicting, in part, the availability of the same resources. Furthermore, because a measurement corresponds to a task, the advisor "sees" only a small subset of the computing environment. If the task requires multiple resources, the measurement conflates their individual availabilities, making it even harder to share measurements. Even when it is possible to untangle the effects of application specifics and the availability of other resources, the resulting measurements of an individual resource are aperiodic and perhaps infrequent. In contrast, the resource-oriented approach measures and predicts resources independently of the application and measures each resource periodically. A resource-oriented advisor can thus make decisions based on up-to-the-minute predictions for all of the available resources. Furthermore, different applications can share these resource predictions just as they share the resources. However, unlike in the application-oriented approach, the advisor must transform these resource predictions into task performance predictions, and this increases the chance of error.

In the explicit resource-oriented prediction approach it is easy and powerful to base prediction on (discrete-time) resource signals, periodic measurements of resource availability. Periodic measurement is easy because measurement is decoupled from the application and instead coupled to the resource. Prediction of a resource signal involves mapping from past values of the signal to future values. This general prediction problem has been and continues to be extensively studied in a number of different fields including statistical signal processing, time series analysis, and chaotic dynamics. By casting the core of the explicit resource-oriented prediction approach as a signal prediction problem, we can bring all of this powerful existing and future machinery to the bear on our scheduling problem. In addition to helping us predict resource signals, these tools also provide a framework for understanding resource availability and for generating meaningful workloads.

Resource signal predictions are not, in themselves, sufficient to solve the scheduling problem posed by real-time scheduling advisors. Such predictions must be reconciled with the resource demands of the task in order to compute a prediction of the running time on which to base scheduling decisions. Our work shows not only that at least one resource signal (CPU availability as measured by host load) can be usefully predicted using statistical signal processing, but also that the gap between these predictions and useful scheduling can indeed be spanned.
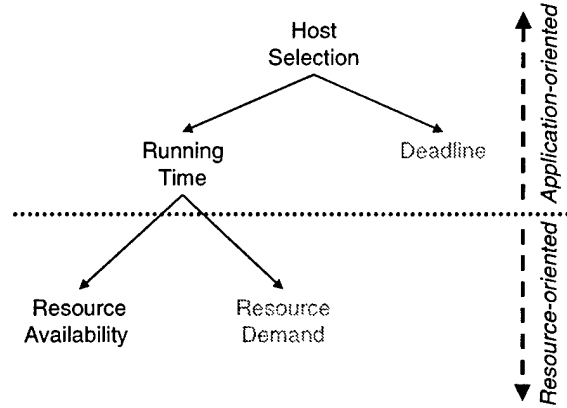
Figure 1.5: Dependencies in a real-time scheduling advisor

1.4.1 Implicit versus explicit prediction

In a design that uses implicit prediction, past measurements of some quantity on each of the prospective hosts are used to order the hosts. The ordering of the hosts with respect to the future performance of the task is assumed to be the same. For example, consider a real-time scheduling advisor based purely on the latest measurement of host load, which we use for comparison purposes in Chapter 6. When the task arrives, the advisor measures the current load on each of the hosts, orders the hosts according to their load, and then assigns the task to the host with the least load, assuming that its running time will be minimized on that host and thus most likely to meet the deadline.

In a design that uses explicit prediction, past measurements are explicitly transformed into predictions of the task's performance and then scheduling decisions are made on the basis of these predictions. The design that forms the core of this dissertation is based on explicit prediction. The system uses statistical signal processing to continuously predict future CPU availability on each of the hosts. The advisor uses these predictions to form statistical estimates of the running time of the task on each of the hosts, and then chooses one of the hosts where the task is likely to meet its deadline with high probability. The measure of task performance need not be the running time. In Appendix A, for example, we look at approaches that use a history of previous successes and failures or a history of previous running times and deadlines to predict whether a deadline can be met on a particular host.

The advantage of implicit approaches is an intrinsic simplicity, since the hosts are merely ordered. In contrast, explicit approaches require computing a prediction of some metric of task performance. However, this value is not only useful to the advisor, but can also be of use to the application, especially if it indicates that the deadline can not be met because insufficient resources are available. In such a case, the application can adjust the resource demands of the task or its deadline to a more realistic level. Another advantage of explicit approaches is that by clearly specifying a prediction problem, the vast machinery of the statistics, signal processing, and artificial intelligence communities becomes available to answer it.

1.4.2 Application-oriented versus resource-oriented prediction

To understand the difference between the explicit application-oriented and resource-oriented prediction approaches, it is useful to consider the dependencies involved in choosing the appropriate host to run a task, as shown in Figure 1.5. The appropriate host depends on the deadline ($t_{now} + (1 + slack)t_{nom}$) and the running time of the task on each of the hosts. The running time of a task on a particular host depends in turn on the resource demand of the task (t_{nom}) and the availability of the resources needed to run the task on that

host (the predicted host load).

Conceptually, the real-time scheduling advisor wants to compute the "host selection" node of the tree before the "resource availability" node is available. In order to do so, it introduces prediction into some node of the dependence tree. The predictive node uses its previous values to predict its current value. This predicted value is then propagated to all its dependencies, and so on. For example, we could introduce prediction at the "resource availability" node by keeping a history of the availability of some host. We would then propagate this prediction upwards to compute a predicted running time on that host, and then use that running time prediction to choose an appropriate host. The other extreme possibility would be to predict at the "host selection" node, basing our choices on how previous host selections fared.

Introducing prediction becomes increasingly difficult the further down the tree we go. The problem is that to propagate a predicted value upwards through tree requires that we be able to model each of the transformations along the way. For example, if we predict at the "host selection" level, we can use our predictions directly. On the other hand, if we predict at the "resource availability" level, we must transform these resource predictions into running time predictions, and then use the predicted running times to predict which host is the most appropriate.

There are several advantages to predicting lower in the tree. First of all, the prediction provides more detail to the application. For example, predicting at the "running time" level or below tells us not only which host is most appropriate for the task, but also what the running time will likely be. This gives the application the opportunity to modify the task's resource demand or deadline before the task is even started. Another advantage is that as we go lower in the tree, the predictions become useful to an increasingly broad range of tools. Predictions at the "host selection" level are only interesting to real-time scheduling advisors. On the other hand, predictions of running time are interesting to scheduling advisors with other goals than real-time. Of course, one could imagine transforming predictions high in the tree into values lower in the tree in order to provide such information. However, notice that each transformation as we move upwards in the tree reduces the amount of information. This means that the transformations can not be uniquely reversible and so information gained by by reversing them can not be as accurate as when measured or predicted directly. Avoiding entanglements by predicting deeper in the tree extends down to individual resources. Predictions of individual resources are easiest to share among applications. The likelihood that two applications will be interested in the same individual resource is much higher than that of them being interested in the same group of resources. A third advantage is that measurements deeper in the tree have the potential to be fresher.

The most important distinction to be made on the basis of Figure 1.5 is between application-oriented and resource-oriented prediction. In application-oriented prediction, which corresponds to the host selection and running time levels, each task execution contributes a single measurement about a single set of resources to the prediction system. This means that the number of sets of resources that are measured, and the frequency with which they are measured is limited by the application and the user. Furthermore, unless each task uses only a single resource, the measurement entangles the availability of multiple resources. Even if each task only uses a single resource or if it is possible to untangle multiple resources, from the point of view of a single resource, measurements would not be periodic, This complicates considerably the use of statistical machinery such as linear time series models. The primary advantage of the application-oriented approach is that the measurements are of quantities that are closer to the metrics that the application is actually concerned about.

In contrast, in resource-oriented prediction, prediction happens at a considerable remove from the application, requiring the development of substantial transformations to predict application-level quantities. In return, however, measurement and prediction can happen independently of applications and the results can be easily shared by multiple applications. Furthermore, resources can be measured and predicted periodically, which easily permits the use of most prediction techniques. Finally, the resource-oriented real-time scheduling advisor has current information available for each of the resources it considers using.

Although this dissertation focuses on the resource-oriented approach, we started our research focused on the application-oriented approach. The results of some of that work are presented in Appendix A. With the appropriate prediction algorithm, we found that application-oriented prediction can be very effective in limited cases, namely those where the nominal time of the tasks is fixed and thus implicitly untangled from CPU availability.

Several issues induced our switch to the resource-oriented approach. Consider the compute-bound case, where a task corresponds to the measurement of a single resource, CPU availability on one host. The first issue was scalability in terms of measurement histories. Consider N applications running on an M host environment. In the application-oriented approach, each application would maintain its own shared measurement history for every host. This means that the amount of measurement history in the system scales as $O(NM)$. When the advisor is run in a system with such a shared measurement history, it needs to collect these measurement histories, resulting in a large amount of communication. Of course, each node could maintain its own local set of histories to avoid the communication, but then the amount of history scales as $O(NM^2)$, which is even worse. In contrast, in the resource-oriented approach the measurement history in the system scales as $O(M)$.

A second issue that argues against the application-oriented approach is entanglement. As we noted earlier, information is reduced as we climb the dependence tree. Consider the running time level. The running time entangles the resource availability (eg, host load), which is a quantity we could share between applications, with the resource demand (eg, nominal time). The result is that if we measure the running time of a task which has some nominal time, this value is only directly applicable to other tasks with the same nominal time. To make it applicable to other tasks requires that we "factor out" the effect of the nominal time. We found this to be non-trivial in practice. With entanglement, the size of the history grows once again. In contrast, the resource-oriented approach never requires this sort of reverse computation, and we found that the forward computations were possible to do accurately.

A third issue is that the lack of periodicity in the measurements of the application-oriented approach severely restricted the kinds of statistical machinery that we could apply to prediction. At the same time, we discovered that host load, measured as a periodically sampled signal, exhibited properties that strongly suggested the use of techniques that rely on periodic measurements. By assuming the periodic measurements possible in the resource-oriented approach, we were able to develop a methodology, called the resource signal methodology (Section 1.6), that we were able to apply not only to host load, but also to network flow bandwidth (Section 7.3). The resource signal abstration lets us leverage old and new work in the fields of time series analysis [23], statistical signal processing [106, 143], chaotic dynamics [1], artificial intelligence [90], and others.

Of course, signal-based resource prediction is not in itself sufficient to implement a real-time scheduling advisor, because it is necessary to model the running time and host selection portions of Figure 1.5. We found that these elements of the explicit resource-oriented prediction approach were indeed feasible and can perform well.

1.5 Prototype real-time scheduling advisor

The core of this dissertation describes the design, implementation, and evaluation of a prototype real-time scheduling advisor. The prototype advisor schedules compute-bound tasks using explicit prediction of *host load signals*, specifically, the Digital Unix five second load average. The low overhead and high performance of this system demonstrate the power of the explicit resource-oriented prediction approach.

Figure 1.6 shows the architecture of the system. At the highest level, the system is divided into two parts: a library, which is bound to a single application, and a daemon, one of which runs indendently on each host and can serve multiple applications. The library and the daemon can be further decomposed into
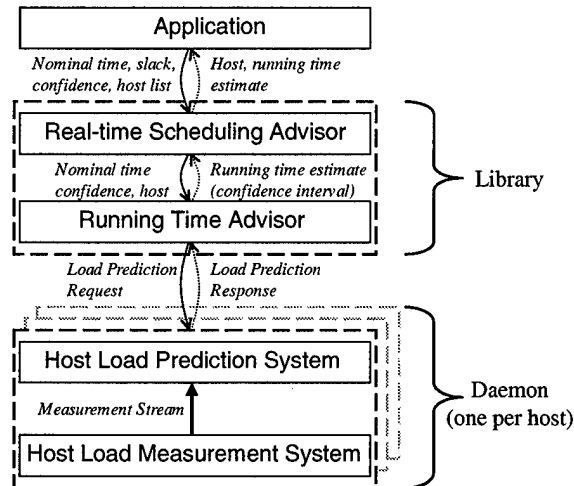
Figure 1.6: The structure of the resource-prediction-based real-time scheduling advisor

independent components that can communicate in a number of ways. In the figure, a dashed arrow represents stream-oriented communication between components, while a symmetric pair of arrows represents request-response communication.

The daemon consists of a host load measurement system, which periodically measures host load, and a host load prediction system, which transforms each new measurement into a qualified prediction of future measurements. Each of the values in the prediction is qualified by an estimate of its error and how it correlates with the error of the other values. The prediction system uses linear time series models to predict host load. It continuously monitors the prediction accuracy, refitting the model when the accuracy drops below a threshold. The daemon stores a short history of the predictions and makes them available via a request-response protocol. This matches the periodic nature of the measurement and prediction systems with the aperiodic nature of application requests.

The library is easiest to describe from the perspective of an application request. A request consists of a nominal running time (t_{nom}), a maximum slack $(slack)$, a confidence level, and a list of hosts. The confidence level, which ranges from zero to one, tells the real-time scheduling advisor the minimum probability of meeting the deadline that the application requires. The real-time scheduling advisor transforms this single application request into requests for predictions of the task's running time on each of the hosts. Each of these requests consists of the nominal time and the confidence level. To answer a request, the running time advisor acquires the latest host load prediction from the host's daemon. It then uses a statistical model of the host's scheduler to transform the host load prediction, the nominal time of the task, and the confidence level into a prediction of the running time of the task on the host. The prediction contains both an expected running time and a confidence interval for the running time. After acquiring running time predictions for each of the hosts, the real-time scheduling advisor chooses a host at random from among those hosts whose running time predictions are less than the deadline. If no such host exists, it chooses the host with the minimum expected running time. The chosen host and the prediction of the running time of the task on that host are returned to the application. The application can then choose whether to accept the solution or to pose a different request.

The application or other middleware services can also use the system at lower levels. For example, other kinds of schedulers can be based on the running time advisor, or can use host load predictions directly. The system is also extensible in a number of ways. It is easy to construct prediction systems for other kinds of resources, and it is easy to add support for new predictive models. The components of the system can also

be arranged differently, using various transports to communicate over the network.

1.6 Resource signal methodology

In addition to arguing for basing real-time scheduling advisors on explicit resource-oriented prediction, this dissertation also recommends a methodology for investigating and implementing such prediction. The methodology, which is described in more detail in Chapter 2, and used in Chapters 3—4, is essentially to transform a specific resource prediction problem into a general time series prediction problem as early as possible, and then to apply the substantial statistical machinery that already exists to address such problems.

The resource signal methodology consists of six steps. First, the investigator finds an easily measured resource signal that correlates with the availability of the resource in question. Second, he uses sampling theory to determine how often the signal needs to be sampled to capture its behavior. Third, he collects representative traces of the sampled signal. These steps require the expertise of a domain expert—a systems researcher. However, the collected traces represent a general time series analysis and prediction problem, for which a large base of expertise and numerous experts already exists. The fourth step is to use the traces to determine the salient statistical properties of the resource signal. This leads to the selection of a set of prospective modeling and prediction techniques. A large number of tools are commercially available to assist with this step. In the fifth step, the investigator performs a randomized evaluation of the prospective models on his traces to determine which model is indeed the most appropriate. In the sixth step, the appropriate model is incorporated into an on-line prediction system for the resource. Few tools are available to help with these latter two steps. We contribute the RPS Toolkit (Chapter 2) to facilitate them. RPS provides tools for carrying out the randomized evaluation, and for rapidly implementing a prediction system based on the appropriate predictive model.

1.7 Outline of dissertation

The flow of the dissertation essentially follows the architecture shown in Figure 1.6, from bottom to top.

Chapter 2 describes the design, implementation, and performance evaluation of the RPS Toolkit, which forms the basis of the host load measurement and prediction systems. RPS provides extensible sensor, prediction, and communication libraries for building resource prediction systems, and a set of components that can be composed at run-time to form resource prediction systems. For the predictive models that we later find appropriate for host load prediction, RPS has extremely low overhead. We also describe an RPS-based parallel evaluation system that we use later to determine the appropriate models for host load prediction. The combination of a powerful off-line evaluation tool and tools for quickly constructing on-line prediction systems based on the evaluation results helps to carry out the resource signal methodology.

Chapter 3 motivates the choice of host load as our resource signal, shows how to appropriately sample it, and describes the statistical characteristics of this resource signal. This knowledge forms the basis for the host load measurement system in Figure 1.6. The interesting and new statistical results are a strong autocorrelation structure, self-similarity, and epochal behavior. These findings suggest that some form of linear model should be appropriate for prediction, but that more complex models that capture long-range dependence may be necessary. Furthermore, they suggest that such models may need to be refitted at epoch boundaries.

Chapter 4 describes a large scale study that evaluated linear models to determine which are most appropriate for host load prediction. The study was based on running randomized testcases using the load traces described in Chapter 3, and data-mining the results. We found that despite the complex behavior of host load signals, relatively simple and computationally inexpensive autoregressive models, of sufficiently high order,

are appropriate for host load prediction. This new knowledge forms the basis for the host load prediction system in Figure 1.6. It was simple to construct this component using RPS once the choice of model became clear.

Chapter 5 describes the design, implementation, and evaluation of the running time advisor component of Figure 1.6. Surprisingly, this component is non-trivial, but we were able to develop an algorithm for computing confidence intervals for the running time. The algorithm relies on two new techniques: accounting for the correlation of prediction errors, and load discounting. We evaluate an implementation of the algorithm by running randomized testcases on real hosts whose workloads are provided by our load traces using a new technique called load trace playback. Essentially, this evaluation tests the bottom three stages shown in Figure 1.6. The results are that the algorithm performs quite well using the autoregressive model we found appropriate for host load prediction. The confidence intervals computed using that model have nearly the desired coverage and are usually far narrower than those computed using other predictive models.

Chapter 6 describes the design, implementation, and evaluation of the real-time scheduling advisor component in Figure 1.6. This component was relatively easy to implement given a functional running time advisor. We evaluate it by running randomized testcases on real hosts whose workloads are generated using load trace playback. We compare our system with simple approaches such as random scheduling and scheduling a task on the host with minimum measured load. Both our system and the measurement approach are vastly superior to random scheduling in terms of the fraction of deadlines that are met. Our system always performs at least as well as the measurement approach and significantly outperforms it in several important regions of operation. Furthermore, unlike the measurement approach, our system is able to tell the application, with very high accuracy, whether the deadline can actually be met on the selected host. This makes it possible for the application to modify the task's requirements until its deadline can be met. Finally, our system is able to introduce appropriate randomness into its scheduling decisions, reducing the chance of synchronization among multiple independent scheduling advisors. Given that these advantages come at very little additional cost over the measurement approach, the superiority of our system is clear.

Chapter 7 concludes the dissertation by describing how our work relates to other work in this area. It also describes the future directions of our work. One direction is to statistically characterize and predict other resources. To this end, we describe an RPS-based prediction system we have developed for network bandwidth, measured using the Remos system, and we present some initial results on evaluating linear models for network bandwidth prediction. Another direction is to develop and incorporate more sophisticated predictive models, which seem to be needed for resource signals such as network bandwidth. We present some initial results on applying a non-linear modeling technique to network bandwidth prediction. Improved modeling of different resource schedulers is another of the directions we contemplate.

Appendix A describes an evaluation of application-oriented prediction approaches.

Chapter 2

Resource Signal Methodology and RPS Toolkit

This dissertation argues for basing real-time scheduling advisors on explicit resource-oriented prediction, specifically on the prediction of resource signals. Implementing such prediction requires a methodology and tools for finding appropriate predictive models and for constructing fast, low overhead on-line prediction systems using the models. This chapter describes our resource signal methodology, and the design, implementation, and performance evaluation of the RPS Toolkit which we have developed to help carry it out. Subsequent chapters apply the methodology and RPS to host load prediction, and show how to use the use the resulting predictions to predict the running time of tasks, resulting in the running time advisor. The predictions of this system then form the basis for our real-time scheduling advisor.

To understand the role RPS plays, consider the process of building a prediction system for a new kind of resource. Once a sensor mechanism has been chosen, this entails essentially two steps. The first is an off-line process consisting of analyzing representative measurement traces, choosing candidate predictive models based on the analysis, and evaluating these models using the traces. The second step is to build an on-line prediction system that implements the most appropriate model with minimal overhead. There are a wide variety of statistical and signal processing tools for interactive analysis of measurement traces that work very well for performing most of the first step. However, tools for doing large scale trace-based model evaluation are usually ad hoc and do not take advantage of the available parallelism. With regard to the second step, building an on-line predictive system using the appropriate model, RPS provides tools for quickly building a on-line resource prediction system out of communicating components.

RPS is designed to be generic, extensible, distributable, portable, and efficient. The basic abstraction is the prediction of periodically sampled, scalar-valued measurement streams, or resource signals. Many such signals arise in a typical distributed system. RPS can easily be extended with new classes of predictive models and new components can be easily implemented using RPS. These components inherit the ability to run on any host in the network and can communicate in powerful ways. The only tool needed to build RPS is a C++ compiler, and it has been ported to four different Unix systems and Windows NT. For typical measurement stream sample rates and predictive models, the additional load that an RPS-based prediction system places on a host is in the noise, while the maximum sample rates possible on a typical machine are as high as 2.7 KHz.

Our experience shows that it is possible to use RPS to find and evaluate appropriate predictive models for a measure of resource availability, and then implement a low overhead prediction system that provides timely and useful predictions for that resource. In Chapter 4, we describe how we used an RPS-based off-line parallel evaluation system to evaluate linear models for host load prediction. Finding that AR(16) models or better are appropriate, we implemented an on-line host load prediction system using this model.

19

The running time and real-time scheduling advisors we describe and evaluate in Chapter 5 and 6 are based on the predictions provided by this system. In this chapter, we evaluate the performance and overheads of this host load prediction system. We have also used RPS to evaluate linear models for network bandwidth prediction, which we describe in Chapter 7. These systems have been used in the CMU Remos [82] resource measurement system, the BBN QuO distributed object quality of service system [145], and are currently being integrated in to the Dv distributed visualization framework [3]. Parts of RPS have also been used to explore the relationship between SNMP network measurements and application-level bandwidth [83].

2.1 Resource signal methodology

To understand the role of RPS, it is useful to consider the task of a systems researcher interested in predicting the availability of a new kind of resource. Many methodologies to attack this problem are possible. However, we are generally interested in how resource availability changes *over time*. It is natural, then, to think in terms of a signal that is related to the resource's availability. In doing so, we recast the problem of predicting resource availability as an abstract signal analysis and prediction problem, which lets us leverage the vast body of research that focuses on such problems. The goal of RPS is to simplify two aspects of this process: the off-line evaluation of predictive models and the construction of on-line systems using appropriate models.

The resource signal methodology that we recommend for our intrepid systems researcher consists of the following steps:

1. Construct a *sensor* for the resource. The sensor generates a periodically sampled, scalar-valued *measurement stream* or *resource signal*.

2. Collect *measurement traces* from representative environments.

3. Analyze these traces using various statistical tools.

4. Choose candidate models based on that analysis.

5. Evaluate the models in an unbiased off-line evaluation study based on the traces to find which of the candidate models are indeed appropriate.

6. Implement an on-line prediction system based on the appropriate models.

The first two steps generally require the researcher to implement custom tools using his domain-specific knowledge of the resource in question. When carrying out the third step, the researcher is aided by powerful, commonly available tools. This step, as well as the fourth step, also relies heavily on the statistical expertise of the researcher. The final two steps can benefit considerably from automated and reusable tools. However, such tools are scarce. The goal of RPS is to provide tools for these last two steps. The researcher should be able to use RPS to conduct off-line model evaluation, and then construct an on-line resource prediction system based on the appropriate models.

The first step requires domain-specific knowledge. There are myriad ways in which interesting signals in a distributed environment can be captured. For example, Section 2.4 describes the OS-specific mechanisms we use to retrieve Unix load averages (average run queue lengths. The Network Weather Service uses benchmarking to measure network bandwidths and latencies between hosts [140] and load average and accounting-based sensors for host load [141]. Remos uses SNMP queries to measure bandwidths and latencies on supported LANs [82].

Two issues that arise in the first step are universal. The sensor implements a sampling process which converts a continuous-time signal into a discrete-time signal. The sampling rate of such a process must be at

least twice the bandwidth of the signal in order to correctly capture it. If a lower sampling rate is desired, the underlying signal must be low-pass filtered to eliminate frequencies greater than half the sampling rate [100, pp. 519]. The second issue is how to achieve periodicity. Aperiodic sampling is simply unavoidable in measuring some resource signals. For example, it is unlikely that networking benchmarks on the wide area can be made to sample periodically. It is necessary to resample such signals to periodicity.

Once the issues involved in the sensor are resolved, the researcher must use his expertise to choose a representative set of environments and capture traces from them using the sensor implementation. This second step is also highly dependent on the particulars of the signal and the environments of interest. Chapter 3 describes how we collected traces for one resource signal: host load.

The appropriately sampled traces that result from the second step pose an abstract signal analysis and prediction problem. The next two steps of the methodology attack this problem. These steps, analyzing the collected traces and choosing candidate models based on the analysis, require far less domain-specific knowledge and more general purpose statistical knowledge. Consequently, there are a number of tools available to help the researcher perform these steps. For example, our study of the properties of host load, which we describe in Chapter 3, made extensive use of the exploratory data analysis and time series analysis tools available in S-Plus [84], and in Matlab's [86] System Identification Toolbox [85]. Steps three and four are labor intensive—finding appropriate predictive models requires human expertise at this point in time. However, because the signal analysis and prediction problem is abstract, it can also be "tossed over the wall" to other researchers with the appropriate expertise.

Surprisingly, there are few tools to simplify the fifth step, evaluating the candidate models in an unbiased manner. Time series analysis methodologies, such as that of Box and Jenkins [23], do generally have an evaluation step, but it is very interactive and primarily concerned with the fit of the model and not its predictive power, which is really what is of interest to the distributed computing community. Furthermore, these methodologies often assume that computational resources are scarce, when, in fact, they are currently widely available.

We believe that the appropriate way to to evaluate prospective models is to study their predictive performance when confronted by many randomly selected testcases. Running this plethora of testcases requires considerable computational resources. While it is possible to script tools such as Matlab and S-Plus to do the job, it would be considerably more efficient to have a more specialized tool that could exploit the embarrassing parallelism available here. Furthermore, it would be desirable for the tool to use the same model implementations that would ultimately be used in the on-line prediction system. Section 2.5.4 describes a parallelized RPS-based tool that does just that. Chapter 4 uses this tool to evaluate linear models for host load prediction.

The final step, implementing an on-line prediction system for the signal, requires implementing, or reusing, the model or models that survived the evaluation step and enabling them to communicate with sensors and the applications or middleware that may be interested in the predictions. In addition, mechanisms to evaluate and control a prediction system must be provided. In Section 2.8.1 we show how we use RPS to implement an on-line host load prediction system using the same models we evaluated earlier. The running time and real-time scheduling advisors described in Chapter 5 and 6 are based on this system.

2.2 Requirements

The goal of RPS is to provide a toolkit that simplifies the final two steps of the resource signal methodology, as described above. We also required that our design would provide genericity, extensibility, distributability, portability, and efficiency. These requirements were intended to make RPS as flexible as possible for future research into resource prediction. We address each of these requirements below, noting how our implementation addresses these requirements.

Genericity: Nothing in the system should be tied to a specific kind of signal or measurement approach. RPS should be able to operate on periodically sampled, scalar-valued measurement streams from any source. Genericity is important in a research system such as RPS because there are a plethora of different signals in a distributed system whose predictability we would like to study. While our research has focused primarily on the prediction of host load as measured by the load average, we have also used RPS to study the predictability of network bandwidth as measured by Remos and through commonly available traces.

Extensibility: It should be easy to add new models to RPS and to write new RPS-based prediction components. Being able to add new models is important because the statistical study of a signal can point to their appropriateness. For example, we added an implementation of the fractional ARIMA model to RPS after we noted that host load traces exhibited self-similarity. An obvious example of the need for easily constructible components is implementing new sensors. For example, we added a Remos-based network bandwidth sensor many months after writing a host load sensor.

Distributability: It should be possible to place RPS-based prediction components in different places on the network and have them communicate using various transports. On-line prediction places computational load on the distributed system and it should be possible to distribute this load across the hosts as desired. Similarly, it should be possible to adjust the communication load and performance by using different transports. For example, many applications may be interested in host load predictions, so it should be easy to multicast them. If two communicating components are located on the same machine, they should be able to use a faster transport. RPS-based components are distributable and can communicate using TCP, UDP, Unix domain sockets, pipes, and files.

Portability: It should be easy to port RPS to new platforms, including those without threads, such as FreeBSD, and non-Unix systems, such as NT. FreeBSD and NetBSD are important target platforms for Remos and RPS. Some of the potential users of RPS such as organizations like the ARPA Quorum project, are increasingly interested in NT-based software. RPS requires only a modern C++ compiler to build and has been ported to Linux, Digital Unix, Solaris, FreeBSD, NetBSD, and Windows NT.

Efficiency: An on-line prediction system implemented using RPS components should be able to operate at reasonably high measurement rates, and place only minor computational and communication loads on the system when operating at typical sampling rates. Obviously, what is reasonable depends on the signal and the model being used to predict it. The idea here is not to achieve near-optimal performance, but rather to achieve sufficient performance to make an RPS-based resource prediction system usable in practice. Ultimately, something like a host load prediction system would probably be implemented as a single, hand-coded daemon which would be considerably faster than a system composed out of communicating RPS prediction components, such as we measure later. However, the latter RPS-based system can operate at over 700 Hz and offers noise-floor level load at appropriate rates with median prediction latencies in the 2 millisecond range. A comparable monolithic system, composed at compile-time using RPS, can sustain a rate of 2.7 KHz.

2.3 Overall system design

Here we describe the structure of RPS as it relates to the construction of an on-line resource prediction system (step 6 of Section 2.1). In addition, RPS can also be used to implement off-line evaluation systems
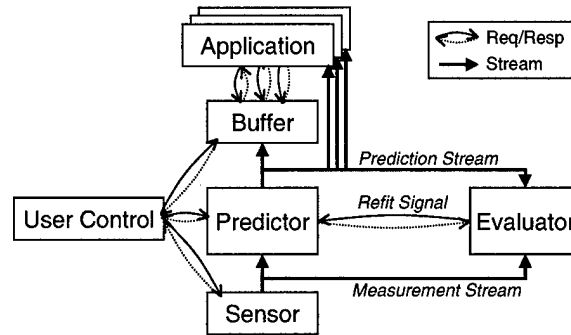
Figure 2.1: Overview of an on-line resource prediction system.

as per step 5 of Section 2.1. In the following, we focus on on-line prediction, pointing out the particulars of off-line prediction only in passing. Section 2.5.4 presents more details about a parallel off-line system.

Figure 2.1 presents an overview of an on-line time series prediction system. In the system, a *sensor* produces a *measurement stream* (we also refer to this as a *signal*) by periodically sampling some attribute of the distributed system and presenting it as a scalar. The measurement stream is the input of a *predictor*, which, for each individual measurement produces a vector-valued prediction. The vector contains predictions for the next m values of the measurement stream, where m is configurable. Each of the predictions in a vector is annotated with an estimate of its error. Consecutive vectors form a *prediction stream*, which is the output of the predictor. *Applications* (including other middleware) can subscribe directly to the prediction stream. The prediction stream also flows into a *buffer*, which keeps short history of the prediction stream and permits applications to access these predictions asynchronously, via a request/response mechanism. The measurement and prediction streams also feed an optional *evaluator*, which continuously monitors the performance of the predictor by comparing the predictor's actual prediction error with a maximum permitted error and by comparing the predictor's estimates of its error with another maximum permitted error level. If either maximum is exceeded—the predictor is either making too many errors or is misestimating its own error—the evaluator calls back to the predictor to tell it to refit its model. The user can exert control of the system by an asynchronous request/response mechanism. For example, he might change the sampling rate of the sensor, the model the predictor is using, or the size of the buffer's history.

The implementation of Figure 2.1 relies on several functionally distinct pieces of software: the sensor libraries, the time series prediction library, the mirror communication template library, the prediction components, and scripts and other ancillary codes.

Sensor libraries implement function calls that measure some underlying signal and return a scalar. Section 2.4 provides more information about host load and flow bandwidth libraries that we provide.

The **time series prediction library** provides an extensible, object-oriented C++ abstraction for time series prediction software as well as implementations of a variety of useful linear models. Section 2.5 provides a detailed description of this library and a study of the stand-alone performance of the various models we implemented.

The **mirror communication template library** provides C++ template classes that implement the communication represented by arrows in Figure 2.1. It makes it very easy to create a component, such as predictor, or any of the other boxes in the figure, which has a large amount of flexibility. In particular, the library provides run-time configurability, the ability to handle multiple data sources and targets, request/response interactions, and the ability to operate over a variety of transports. Section 2.6 describes the mirror library in detail.

Prediction components are programs that we implemented using the preceding libraries. They realize the boxes of Figure 2.1 and can be connected as desired when they are run. Section 2.7 describes the

prediction components we implemented. Section 2.8 describes the performance and overhead of an on-line host load prediction system composed from these components and communicating using TCP.

Ancillary software includes scripts to instantiate prediction systems on machines, tools for replaying host load traces, and tools for testing host load prediction-based schedulers. We don't describe this software in any further detail in this chapter. One piece of software that has not been implemented is a system for keeping track of instantiated prediction systems and their underlying data streams. Currently, the client middleware or the user must instantiate prediction components and manage them.

2.4 Sensor libraries

Currently, two sensor libraries have been implemented. The first library, *GetLoadAvg* provides a function that retrieves the load averages (ie, average run queue length) of the Unix system it is running on. On some systems, such as Digital Unix, these are 5, 30, and 60 second averages, while on others, such as Linux, these are 1, 5, and 15 minute averages. As we shall show in Chapter 3, the running time of a compute-bound task on a Digital Unix system is strongly related to the average load it experiences during execution.

The GetLoadAvg code was borrowed from Xemacs and considerably modified. It uses efficient OS-specific mechanisms to retrieve the numbers where possible. When such mechanisms are not available, or when the user's permissions are inadequate, it runs the Unix uptime utility and parses its output. Because NT does not have an equivalent to the load average, it gracefully fails on that platform. Additional code is available from us for directly sampling the run-queue length on an NT system using the registry interface. On a 500 MHz Digital Unix machine, approximately 640,000 GetLoadAvg calls can be made per second. The maximum observed latency is about 10 milliseconds. We normally operate at about 1 call per second.

The second library, *GetFlowBW*, provides a function that measures the bandwidth that a prospective new flow between two IP addresses would receive, assuming no change in other flows. The implementation is based on Remos [82], which uses SNMP queries to estimate this value on LANs and benchmarking to estimate it on WANs. For SNMP queries on a private LAN, about 14 calls can be made per second.

2.5 Time series prediction library

The time series prediction library is an extensible set of C++ classes that cooperate to fit models to data, create predictors from fitted models, and then evaluate those predictors as they are used. While the abstractions of the library are designed to facilitate on-line prediction, we have also implemented several off-line prediction tools using the library, including a parallelized cross-validation tool. Currently, the library implements the Box-Jenkins linear time series models (AR, MA, ARMA, ARIMA), a fractionally integrated ARIMA model which is useful for modeling long-range dependence dependence such as arises from self-similar signals, a "last value" model, a windowed average model, and a long term average model. In addition, we implemented a template-based utility model which can be parameterized with another model resulting in a version of the underlying model that periodically refits itself to data.

2.5.1 Abstractions

The abstractions of the time series prediction library are illustrated in Figure 2.2. The resource signal consists of the values $\langle z_t \rangle = \langle z_{t-\infty}, \ldots, z_{t-1}, z_t, z_{t+1}, \ldots, z_{t+\infty} \rangle$, where t is the current time. The user begins with a *measurement sequence*, $\langle z_{t-N}, \ldots, z_{t-2}, z_{t-1} \rangle$, which is a sequence of N previous scalar values that were collected at periodic intervals, and a *model template* which contains information about the structure of the desired model. Although the user can create a model template himself, a function is also provided to create a model template by parsing a sequence of strings such as command-line arguments.
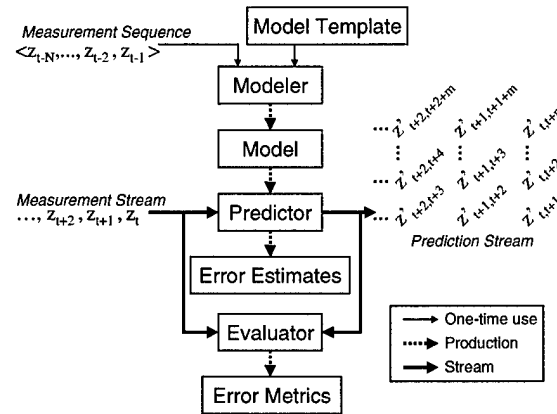
Figure 2.2: Abstractions of the time series prediction library.

The measurement sequence and model template are supplied to a *modeler* which will fit a *model* of the appropriate structure to the sequence and return the model to the user. The user can select the appropriate modeler himself or use a provided function which chooses it based on the model template. The returned model represents a fit of the model structure described in the model template to the measurement sequence.

To predict future values of the resource signal, the model creates a *predictor*. A predictor is a filter which operates on a scalar-valued *measurement stream* of signal samples as they arrive, $z_t, z_{t+1}, \ldots$, producing a vector-valued *prediction stream*, $[\hat{z}_{t,t+1}, \hat{z}_{t,t+2}, \ldots, \hat{z}_{t,t+m}], [\hat{z}_{t+1,t+2}, \hat{z}_{t+1,t+3}, \ldots, \hat{z}_{t+1,t+1+m}], \ldots$ Each new measurement generates predictions for what the next m measurements will be, conditioned on the fitted model and on all the measurements up to and including the new measurement. m can be different for each step and the predictor can be asked for any arbitrary next m values at any point. The predictor can also produce *error estimates* for its $1, 2, \ldots, m$-step ahead predictions. For the linear models used in this dissertation, an error estimate is a covariance matrix of the expected prediction errors. Capturing the relationship between the different prediction errors is important, as we discuss in Chapter 5. Ideally, the prediction errors will be normally distributed and so these estimates can serve to compute a confidence interval for the prediction.

The measurement and prediction streams can also be supplied to an *evaluator*, which evaluates the actual quality of the predictions independent of any particular predictor, producing *error metrics*. The user can compare the evaluator's error metrics and the predictor's error estimates to determine whether a new model needs to be fitted.

2.5.2 Implementation

The time series prediction library is implemented in C++. To extend the basic framework shown in Figure 2.2 to implement a new model, one creates subclasses of model template, modeler, model and predictor, and updates several helper functions. We implemented the nine different predictive Models and their corresponding Modelers. Most of these models share a single Predictor implementation. All the models share a single Evaluator implementation.

Models and modelers

The models we implemented are different kinds of linear time series models. The main idea behind linear time series models is to treat the signal, $\langle z_t \rangle$, as a realization of a stochastic process that can be modeled as a white noise source driving a linear filter. The filter coefficients can be estimated from past observations of
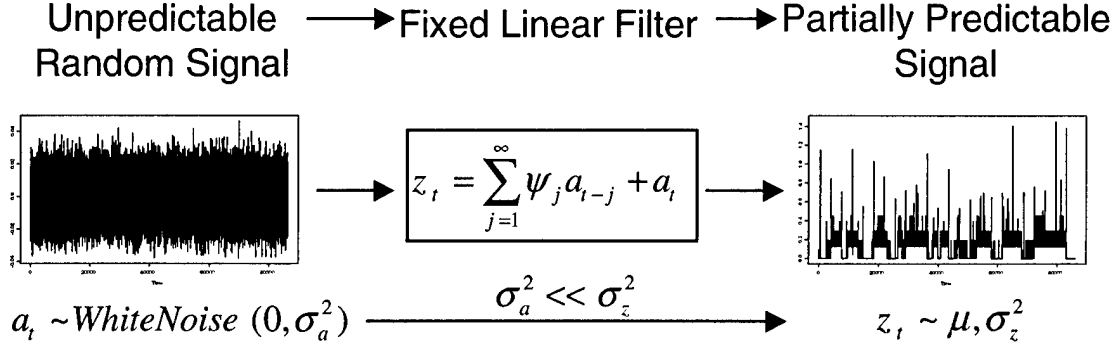
Figure 2.3: Linear time series model.

Model	Notes
Simple Models	
MEAN	Long-range mean
LAST	Last-value
BM(p)	Mean over "best" window
Box-Jenkins Models	
AR(p)	Uses Yule-Walker
MA(q)	Uses Powell
ARMA(p,q)	Uses Powell
ARIMA(p,d,q)	Captures non-stationarity, uses Powell
Self-similar Models	
ARFIMA(p,d,q)	Captures long-range dependence
Utility Models	
REFIT<T>	Auto-refitting model

Table 2.1: Currently implemented predictive models.

the signal, namely the measurement sequence $\langle z_{t-N}, \ldots, z_{t-2}, z_{t-1}\rangle$ of Figure 2.2. If most of the variability of the signal results from the action of the filter, we can use its coefficients to estimate future signal values with low mean squared error.

Figure 2.3 illustrates this decomposition. In keeping with the relatively standard Box-Jenkins notation [23], we represent the input white noise signal as $\langle a_t \rangle$ and the output signal as $\langle z_t \rangle$. On the right of Figure 2.3 we see our partially predictable signal $\langle z_t \rangle$, which exhibits some mean μ and variance σ_z^2. On the left, we see our utterly unpredictable white noise signal $\langle a_t \rangle$, which exhibits a zero mean and a variance σ_a^2. In the middle, we have our fixed linear filter with coefficients $\langle \psi_j \rangle$. Each output value z_t is the sum of the current noise input a_t and all previous noise inputs, weighted by the $\langle \psi_j \rangle$ coefficients.

Given an observed output signal $\langle z_t \rangle$, the optimum values for the coefficients ψ_j are those that minimize σ_a^2, the variance of the driving white noise signal $\langle a_t \rangle$. Notice that the one-step-ahead prediction given all the data up to and including time t is $\hat{z}_{t,t+1} = \sum_{j=0}^{\infty} \psi_j a_{t-j}$, since the expected value of $a_{t+1} = 0$). The noise signal consists simply of the one-step-ahead prediction errors and the optimal coefficient values minimize the sum of squares of these prediction errors. In practice, the coefficients of the linear model are estimated from a subset of the signal, namely the measurement sequence of figure 2.2. If the characteristics of the signal change over time, then it will become necessary to refit the model.

The general form of the linear time series model is, of course, impractical, since it involves an infinite summation using an infinite number of completely independent weights. Practical linear time series models

use a small number of coefficients to represent infinite summations with restrictions on the weights, as well as special casing the mean value of the signal, μ. To understand these models, it is easiest to represent the weighted summation as a ratio of polynomials in B, the backshift operator, where $B^d z_t = z_{t-d}$. For example, we can write $z_t = \sum_{j=1}^{\infty} \psi_j a_{t-j} + a_t$ as $z_t = \psi(B)a_t$ where $\psi(B) = 1 + \psi_1 B + \psi_2 B + \ldots$ Using this scheme, the models we examine in this chapter can be represented as

$$z_t = \frac{\theta(B)}{\phi(B)(1-B)^d} a_t + \mu \tag{2.1}$$

where the different classes of models we examine in this chapter (AR, MA, ARMA, ARIMA, ARFIMA, BM, and MEAN) constrain $\theta(B)$, $\phi(B)$ and d in different ways. In the signal processing domain, this kind of filter is known as a pole-zero filter. The roots of $\theta(B)$ are the zeros and the roots of $\phi(B)(1-B)^d$ are the poles. In general, such a filter can be unstable in that its outputs can rapidly diverge from the input signal. This instability is extremely important from the point of view of the implementor of a resource prediction system. Such a system will generally fit a model (choose the $\theta(B)$ and $\phi(B)$ coefficients, and d) using some n previous observations. The model will then be "fed" the next m observations and asked to make predictions in the process. If coefficients are such that the filter is unstable, then it may explain the n observations very well, yet fail miserably and even diverge (and crash!) when used on the m observations after the fitting.

MEAN model: The MEAN model has $z_t = \mu$, so all future values of the signal are predicted to be the mean. This is the best predictor, in terms of minimum mean squared error, for a signal which has no correlation over time—in other words, it is best if the signal is entirely white noise. The MEAN modeler and model classes essentially do no work, while the MEAN predictor class maintains a running estimate of the mean and variance of the signal.

LAST model: LAST models have $z_t = \frac{1}{\phi(B)} a_t$ where $\phi(B)$ has one coefficient, set to one. In other words, $z_t = z_{t-1}$, so all future values are predicted to be the same as the last measured value. LAST is implemented as a BM(1) model, which we describe next.

BM(p) models: BM(p) models have $z_t = \frac{1}{\phi(B)} a_t$ where the $\phi(B)$ has N, $N \leq p$, coefficients, each set to $1/N$. This simply predicts the next signal value to be the average of the previous N values, a simple windowed mean. The BM(p) modeler chooses N to minimize the one-step-ahead prediction error for the measurement sequence. The BM(p) model simply keeps track of this N and the BM(p) predictor implements the windowed average.

AR(p) models: AR(p) models (purely autoregressive models) have $z_t = \frac{1}{\phi(B)} a_t + \mu$ where $\phi(B)$ has p coefficients. Intuitively, the output value is a ϕ-weighted sum of the p previous output values. The $t+1$ prediction for a signal is the ϕ-weighted sum of the p previous measurements. The $t+2$ prediction is the ϕ-weighted sum of the $t+1$ prediction and the $p-1$ previous measurements, and so on.

From the point of view of a system designer, AR(p) models are highly desirable since they can be fit to data in a deterministic amount of time. In the Yule-Walker technique that we used, the autocorrelation function is computed to a maximum lag of p and then a p-wide Toeplitz system of linear equations is solved. Even for relatively large values of p, this can be done almost instantaneously.

MA(q) models: MA(q) models (purely moving average models) have $z_t = \theta(B)a_t$ where $\theta(B)$ has q coefficients. Intuitively, the output value is the θ-weighted sum of the current and the q previous input

values. The $t + 1$ prediction of a signal is the θ-weighted sum of the q previous $t + 1$ prediction errors. The $t + 2$ prediction is the θ-weighted sum of the predicted $t + 1$ prediction error (zero) and the $q - 1$ previous $t + 1$ prediction errors, and so on.

MA(q) models are a much more difficult proposition for a system designer since fitting them takes a nondeterministic amount of time. Instead of a linear system, fitting a MA(q) model presents us with a quadratic system. Our implementation, which is nonparametric (ie, it assumes no specific distribution for the white noise source), uses the Powell procedure [105, 406–413] to minimize the sum of squares of the $t+1$ prediction errors. The number of iterations necessary to converge is nondeterministic and data dependent.

ARMA(p,q) models: ARMA(p,q) models (autoregressive moving average models) have $z_t = \frac{\theta(B)}{\phi(B)} a_t + \mu$ where $\phi(B)$ has p coefficients and $\theta(B)$ has q coefficients. Intuitively, the output value is the ϕ-weighted sum of the p previous output values plus the θ-weighted sum of the current and q previous input values. The $t + 1$ prediction for a signal is the ϕ-weighted sum of the p previous measurements plus the θ-weighted sum of the q previous $t + 1$ prediction errors. The $t + 2$ prediction is the ϕ-weighted sum of the $t + 1$ prediction and the $p - 1$ previous measurements plus the θ-weighted sum of the predicted $t + 1$ prediction error (zero) and the $q - 1$ previous prediction errors, and so on.

By combining the AR(p) and MA(q) models, ARMA(p,q) models hope to achieve greater parsimony— using fewer coefficients to explain the same signal. From a system designer's point of view, this may be important, at least in so far as it may be possible to fit a more parsimonious model more quickly. Like MA(q) models, however, ARMA(p,q) models take a nondeterministic amount of time to fit to data, and we use the same Powell minimization procedure to fit them.

ARIMA(p,d,q) models: ARIMA(p,d,q) models (autoregressive integrated moving average models) implement Equation 2.1 for $d = 1, 2, \ldots$ Intuitively, the $(1 - B)^d$ component amounts to a d-fold integration of the output of an ARMA(p,q) model. Although this makes the filter inherently unstable, it allows for modeling nonstationary signals if they change smoothly. Such signals can vary over an infinite range and have no natural mean. Although most resource signals can not vary infinitely, they often don't have a natural mean, either.

ARIMA(p,d,q) models are fit by differencing the measurement sequence d times and fitting an ARMA(p,q) model as above to the result.

ARFIMA(p,d,q) models: ARFIMA(p,d,q) models (autoregressive fractionally integrated moving average models) implement Equation 2.1 for fractional values of d, $0 < d < 0.5$. By analogy to ARIMA(p,d,q) models, ARFIMAs are fractionally integrated ARMA(p,q) models. The details of fractional integration [63, 55] are not important here other than to note that $(1 - B)^d$ for fractional d is an infinite sequence whose coefficients are functions of d. The idea is that this infinite sequence captures long range dependence while the ARMA coefficients capture short range dependence [15]. As we note in Chapter 3, host load exhibits long-range dependence, even after differencing, thus ARFIMAs may prove to be beneficial models.

To fit ARFIMA models, we use Fraley's Fortran 77 code [49], which does maximum likelihood estimation of ARFIMA models following Haslett and Raftery [61]. This implementation is also used by commercial packages such as S-Plus. We truncate $(1 - B)^d$ at 300 coefficients and use the same representation and prediction engine as with the other models.

REFIT<T> model: The REFIT<T> modeler, model, and predictor are C++ template classes that are parameterized by some modeler class and produce models of the underlying type that will automatically refit themselves at regular, user-specified, intervals. For example,

```
RefittingModeler<ARModeler>::Fit(seq,seqlen,modeltemplate,interval)
```

will return an AR model whose predictor will automatically fit a new AR model and update itself after every `interval` new samples.

Note on Powell's method: The choice of Powell's method, which we use in our implementations of the MA, ARMA and ARIMA models is a compromise. Powell's method does not require derivatives of the function being minimized, but operates more slowly than other methods which can make use of derivatives.

We use this method because we want to minimize σ_a^2 (the sum of squared prediction errors) directly. Other, faster methods to fit MA, ARMA, and ARIMA models exist. Instead of minimizing σ_a^2, these methods maximize the likelihood, which is a function of σ_a^2 whose form is determined by the distribution of the errors. By *assuming a particular distribution*, a function with known derivatives is produced and this allows the use of faster function minimization methods. However, we found that assuming a particular error distribution was rarely valid for host load and network flow bandwidth, two signals that were of considerable interest to us. The prediction errors of linear time series models on real signals are rarely distributed according to a convenient analytic distribution, although they usually are quite white (uncorrelated) and have low σ_a^2.

Predictors

As we noted earlier, the MEAN, LAST, and BM(p) models have corresponding simple predictors. The AR, MA, ARMA, ARIMA, and ARFIMA models share a single predictor implementation, called the eta-theta predictor. This predictor maintains a copy of the model coefficients $\eta(B) = \phi(B)(1-B)^d$, $\theta(B)$ as before, μ, and σ_a^2. In addition, it maintains prediction state in the form of the predicted next signal value, a queue that holds the last $p + d$ ($d = 300$ for ARFIMA models) measurements, and a queue the holds the last q prediction errors. When the next measurement becomes available, it is pushed onto the measurement queue, its corresponding predicted value's error is pushed onto the error queue, and the model is evaluated ($O(p + d + q)$ operations) to produce a new predicted next signal value. We refer to this as *stepping* the predictor. At any point, any predictor can be queried for the predicted next k values of the signal, along with their expected mean squared errors ($O(k(p + d + q)$ operations)). The user can request the expected mean squared errors for individual predictions (eg, $\hat{z}_{t+1}$), or for the covariance matrix or autocovariance sequence of a sequence of predictions. The covariances are needed if the final result is to be some function of the individual predictions in the sequence.

Evaluator

The evaluator we implemented measures the following error metrics of a predictor. For each lead time, the minimum, median, maximum, mean, mean absolute, and mean squared prediction errors are computed. Of these, the mean squared prediction errors are especially useful, since they can be compared against the predictor's own estimates to determine whether a new model needs to be fitted. Of course, a new model can also be fitted if the prediction error is simply too high, or for any reason, at any time.

The one-step-ahead prediction errors (ie, a_{t+i}^1, $i = 1, 2, \ldots, n$) are also subject to IID and normality tests as described by Brockwell and Davis [25, pp. 34–37]. IID tests include the fraction of the autocorrelations that are significant, the Portmanteau Q statistic (the power of the autocorrelation function), the turning point test, and the sign test. Recall that with an adequate model, the prediction errors should be uncorrelated (white) noise. If an IID test finds significant correlation in the errors, then a new model can be fitted to attempt to capture this correlation. The evaluator also tests if the errors are distributed normally by computing the R^2 value of a least-squares fit to a quantile-quantile plot of the errors versus a sequence of

normals of the same mean and variance. If the R^2 is high, then using the simplifying assumption that the errors are normally distributed is well founded.

2.5.3 Example

The following is a code fragment to show how the time series prediction library can be used. In the code, we fit an ARMA(2,2) model to the first half of the sequence. seq and then do 8-step-ahead predictions on the second half of the sequence.

```
ModelTemplate *template;
Model          *model;
Predictor      *predictor;
Evaluator      *evaluator;
double         predictions[8], errorestimates[8];

// fit model to 1st half and create predictor
template = ParseModel(3,{"ARMA","2","2"});
model    = FitThis(&(seq[0]),seqlen/2,*template);
predictor = model->MakePredictor();
eval     = new Evaluator;

// bring predictor state up to date
Prime(predictor,&(seq[0]),seqlen/2);

evaluator->Initialize(8);

// 8-ahead predictions for rest of sequence
for (i=seqlen/2+1;i<seqlen;i++) {
    // Step the new observation into the predictor - this
    // returns the current one step ahead prediction, but
    // we're just ignoring it here.
    predictor->Step(seq[i]);
    // Ask for predictions + errors from 1 to 8 steps into the future
    // given the state in the predictor at this point
    predictor->Predict(8,predictions);
    predictor->ComputeVariances(8,errorestimates);
    // Send output to evaluator
    evaluator->Step(seq[i],predictions);
    // do something useful with  predictions here
}

// Get final stats from evaluator
evaluator->Drain();
PredictionStats *predstats = evaluator->GetStats();
```

To use a different model, all that is needed is to change the arguments to the ParseModel call, which could just as easily come from the command line. ParseModel and FitThis are helper functions to simplify dealing with the large and extensible set of available model templates, modelers, models, and predictors. It is also possible to invoke modelers directly, with or without model templates.

2.5.4 Parallel cross-validation system

Using the time series prediction library, we implemented a parallel cross-validation system for studying the predictive power of models on traces of measurement data. The user supplies a measurement trace and a file

containing a sequence of testcase templates. A testcase template contains ranges of valid values for model classes, numbers of model parameters, lengths of sequences to fit models to, and lengths of subsequent sequences to test the fitted models on.

As the system runs, testcases are randomly generated by a master program using the template's limits on valid values and parceled out to worker processes using PVM [51]. The workers run code similar to that of Section 2.5.3 to evaluate a testcase. Essentially, the result is a set of error metrics for a randomly chosen model fit to a random section of the trace and tested on a subsequent random section of the trace. When a worker finishes evaluating a testcase, it sends the resulting set of error metrics back to the master, which prints them in a form suitable for importing into a database table for further study.

Because the testcases are randomly generated, the database of testcases can be used to draw unbiased conclusions about the absolute and relative performance of particular prediction models on particular kinds of measurement sequences.

2.5.5 Performance

In implementing an on-line resource prediction system, it is obviously important to know the costs involved in using the various models supported by the time series prediction library. For example, if the measurement stream produces data at a 10 Hz rate and the predictor requires 200 ms to produce a prediction, then it will fall further and further behind, producing "predictions" for times that are increasingly further in the past. Clearly, such a predictor is useless. Another predictor that requires 100 ms will give up-to-date predictions, but at the cost of saturating the CPU of the machine where it is running. A predictor that requires 1 ms or less would clearly be desirable since it would consume only 1% of the CPU. Similarly, the cost of fitting a model and the measurement rate determines how often we can refit the model. At the 10 Hz rate, a model that takes 10 seconds to fit cannot be fit any more often than every 100 measurements, and only then if we are willing to saturate the CPU.

We measured the costs, in terms of system and user time required to (1) fit a model and create a predictor and (2) step one measurement into the predictor producing one set of 30-step-ahead predictions. The machine we used is a 500 MHz Alpha 21164-based DEC personal workstation. Because the time to fit a model is dependent on the length of the measurement sequence, while the prediction time is not, we measured the costs for two different measurement sequence lengths, 600 samples and 2000 samples. The measurement sequence used was a representative host load trace from among those described in Chapter 3.

The results are shown in Figures 2.4 and 2.5. Each figure contains six plots, one for the (a) MEAN, LAST, and AR models, and one each for the remaining (b) BM, (c) MA, (d) ARMA, (e) ARIMA, and (f) ARFIMA models. The REFIT<T> variants were not measured, although their performance can certainly be derived from the measurements we did take. For each model, we plot several different and interesting combinations of parameter values. For each combination, we plot two bars. The first bar (Fit/Init) plots the time to fit the model and produce a predictor, while the second bar (Step/Predict) plots the time to step that predictor. Each bar is the average of 30 trials, each of which consists of one Fit/Init step and a large number of Step/Predict steps. The y axis on each plot is logarithmic. We replicate some of the bars from graph to graph to simplify comparing models across graphs, and we also draw horizontal lines at roughly 1 ms and 100 ms, which are the Fit/Init times of AR(16) and AR(512) models, respectively. 1 ms is also the Step/Predict time of an AR(512) predictor.

There are several important things to note when examining Figures 2.4 and 2.5. First, the inclusion of LAST and MEAN on the (a) plots provide measures of the overhead of the predictor and modeler abstractions, since LAST's predictor and MEAN's modeler do hardly any work. As we can see, the overhead of the abstractions are quite low and on par with virtual function calls, as we might expect.

The second important observation from the figures is that AR models, even with very high order, are

(a) AR Models

(b) BM Models

(c) MA Models

(d) ARMA Models

(e) ARIMA Models

(f) ARFIMA Models
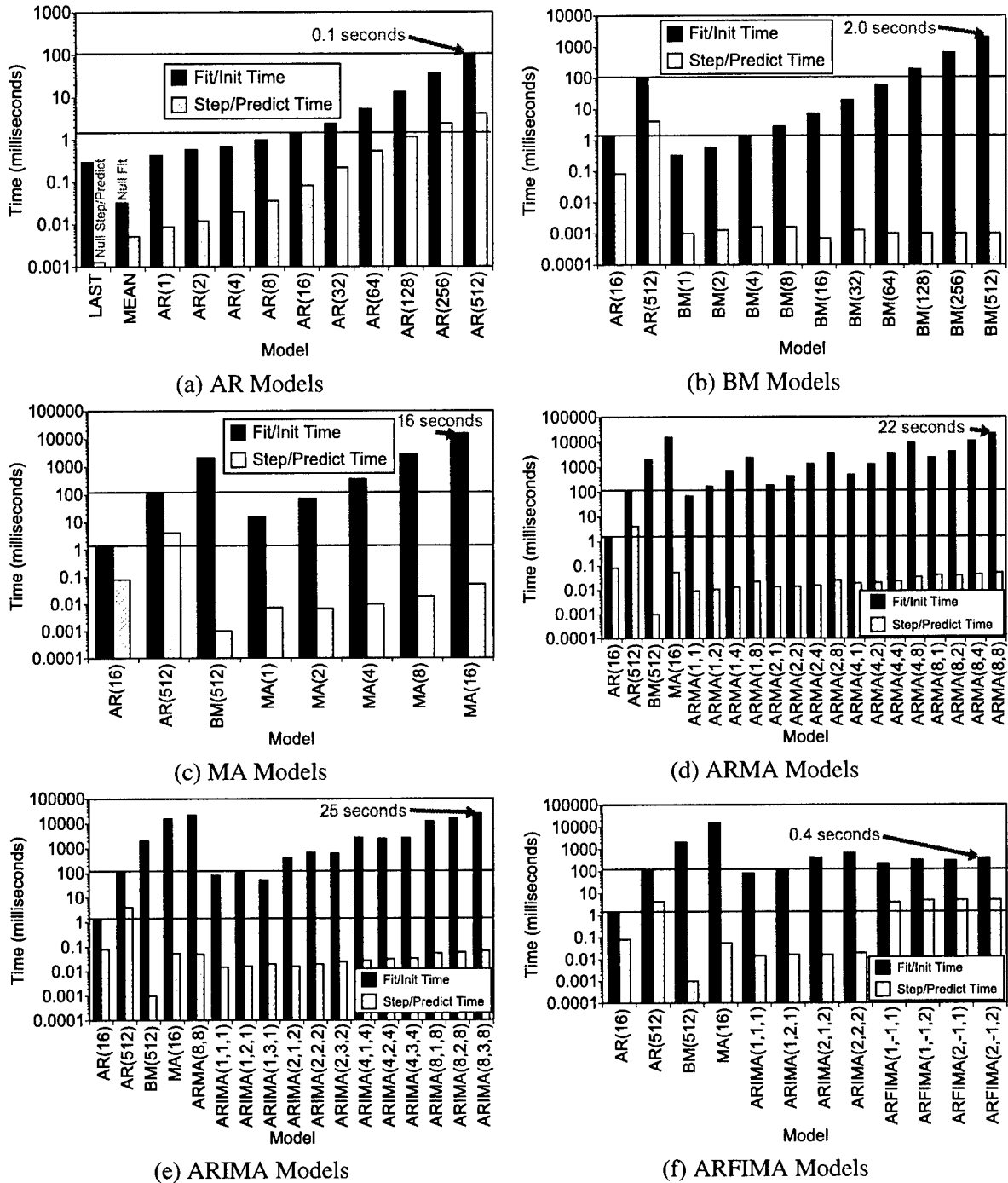
Figure 2.4: Timing of various prediction models, 600 sample fits.

quite inexpensive to fit. An AR(512) fit on a 2000 element sequence takes about 100 ms. In fact, ignoring LAST and MEAN, the only real competition to even the AR(512) comes from very low order versions of the other models. The downside of high order AR models is that the Step/Predict time tends to be much higher than that of lower order versions of the more complex models. For example, the predictor for an ARIMA(8,3,8) model operates in 1/100 the time of an AR(512). This is because the number of operations an eta-theta predictor performs is linear in the number of model parameters. If very high measurement rates

(a) AR Models

(b) BM Models

(c) MA Models

(d) ARMA Models

(e) ARIMA Models
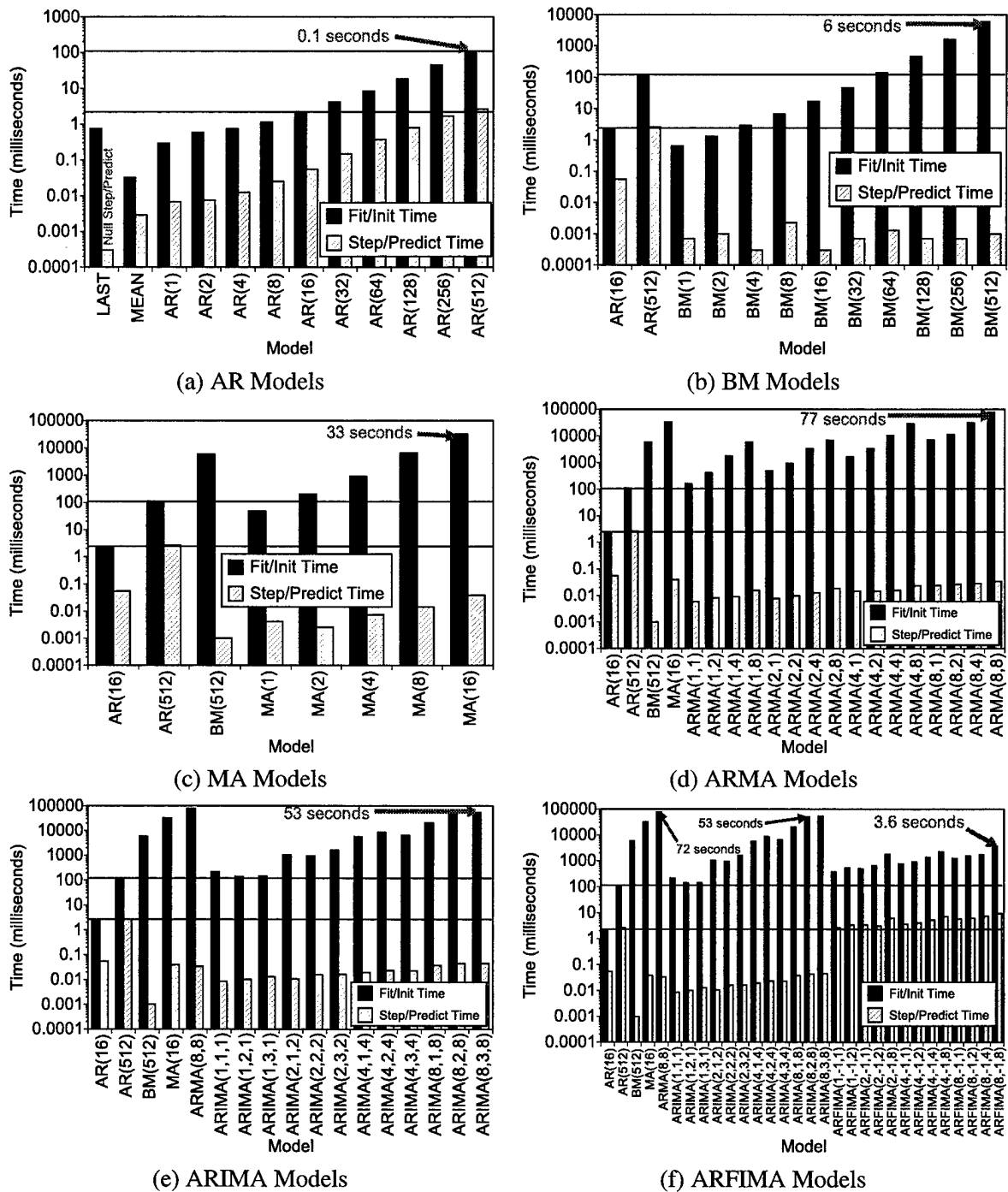
(f) ARFIMA Models

Figure 2.5: Timing of various prediction models, 2000 sample fits.

are important, these more parsimonious models may be preferable. Interestingly, the ARFIMA models also have very expensive predictors. This is because, although the model captures long-range dependence very parsimoniously in the form of the d parameter, we multiply out the $(1-B)^d$ term to generate 300 coefficients in the eta-theta predictor. It is not clear how to avoid this.

A final observation is that the MA, ARMA, and ARIMA models, quite surprisingly, are considerably
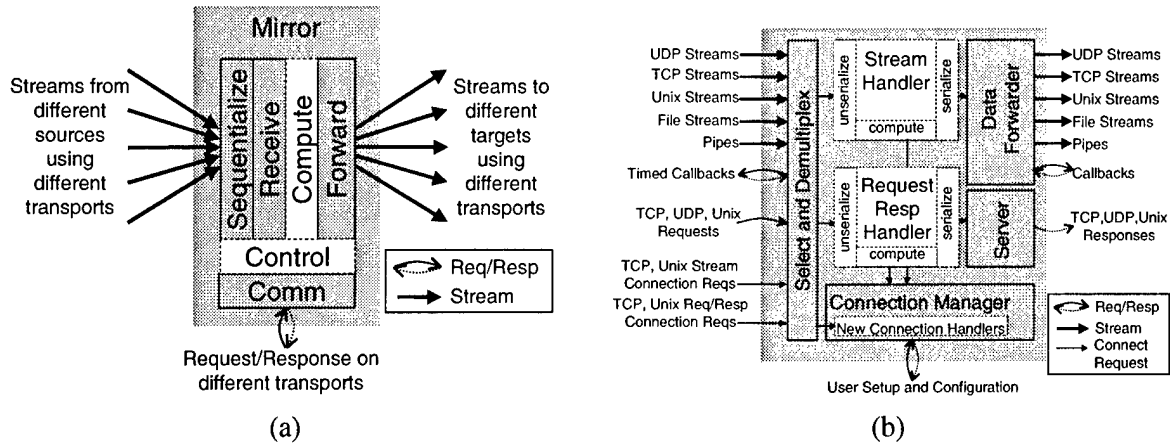
Figure 2.6: The mirror abstraction (a) and implementation (b).

more expensive to fit than the much more complex ARFIMA models. This is because we use a highly-tuned maximum likelihood code that assumes a normal error distribution to fit the ARFIMA model. The MA, ARMA, and ARIMA models are fit without making this assumption using a function optimizer which does not require derivatives. We used this approach because experimentation with Matlab, which uses a maximum likelihood approach, showed that the assumption was rarely valid for traces we were interested in. Maximum likelihood based modelers for MA, ARMA, and ARIMA models would reduce their Fit/Init times to a bit below those of the ARFIMA models. However, fitting even high-order AR models should still be cheaper because $AR(p)$ models are fit by solving a p-diagonal Toeplitz while the other models require some form of function optimization over their parameters.

2.6 Mirror communication template library

As we began to implement an on-line resource prediction service for host load and contemplated implementing another for network bandwidth, we discovered that we were often rewriting the same communication code in each new program. As the number of such prediction components began to grow and we sought to incorporate more sophisticated communication transports such as multicast IP, the situation became untenable. Stepping back, we factored out the communication requirements of the prediction components and decided to implement support for them separately.

2.6.1 Motivation

Consider Figure 2.1, which shows a high level view of how the components of an on-line prediction service communicate. Notice that each component can is roughly similar in how it communicates with other components. It receives data from one or more input *data streams* and sends data to one or more output data streams. When a new data item becomes available on some input data stream, the component performs computation on it and forwards it to all of the output data streams. In addition to this data path the component also provides *request-response control* which operates asynchronously. We refer to this abstraction as a *mirror* (with no computation, input data is "reflected" to all of the outputs) and illustrate it in Figure 2.6(a).

We wanted to be able to implement the communication a mirror performs in different ways depending on where the components are situated and how many there are. For example, the predictor component in Figure 2.1 accepts a stream of measurements from a sensor and produces a stream of predictions which are consumed by the buffer and the evaluator. In addition, the evaluator and the user can asynchronously

request that the predictor refit its model. Applications may connect at any time to receive the prediction stream. If we colocate all of the components on a single machine, Unix domain sockets or even pipes might be the fastest communication mechanism to use. If the components are on different machines—for example, it may be impossible to locate any components other than the sensor on a network router—TCP-based communication may be preferable. If a large number of applications are interested in the prediction stream, it may be necessary to use multicast IP to keep network traffic low. This is the kind of flexibility we expected from our mirror implementation.

2.6.2 Implementation

Our mirror implementation, illustrated in Figure 2.6(b), is a C++ template class which is parameterized at compile-time by handlers for stream input and for request/response input. Additionally, it is parameterized by handlers for new connection arrivals for streams and for request/response traffic, although the default handlers are usually used for this functionality. Parameterized stream input and request-response handlers are also supplied for serializable objects, which can be used to hide all the details of communication from the computation that a mirror performs for data or control. Beyond this, there are other template classes and default handler implementations to simplify using a mirror. For example, our prediction mirror implementation uses one of these templates, `FilterWithControl<>`, to simplify its design:

```
class Measurement : public SerializableInfo {...};
class PredictionResponse : public SerializableInfo {...};
class PredictionReconfigurationRequest : public SerializableInfo {...};
class PredictionReconfigurationReply : public SerializableInfo {...};

class Prediction {
...
public:
  static int Compute(Measurement &measure,
                     PredictionResponse &pred);
...
}

class Reconfiguration {
...
public:
 static int Compute(PredictionReconfigurationRequest &req,
                    PredictionReconfigurationResponse &resp);
...
}

typedef FilterWithControl<
  Measurement,
  Prediction,
  PredictionResponse,
  PredictionReconfigurationRequest,
  Reconfiguration,
  PredictionReconfigurationResponse
> PredictionMirror;
```

To implement serialization, the classes descended from `SerializeableInfo` implement methods for getting their packed size and for packing and unpacking their data to and from a buffer object. The implementer of the prediction mirror does not write any communication code, which is all provided in the `FilterWithControl` template which ultimately expands into a mirror template.

As shown in Figure 2.6(b), the heart of a mirror is a select statement that waits for activity on the file descriptors associated with the various input streams, request/response ports, and ports where new connections arrive. Streams can also originate from in-process sources, and so the select includes a timeout for periodically calling back to these local sources to get input stream data.

When the select falls through, all local callbacks that are past due are executed and their corresponding stream handler is executed on the new data item. Next, each open file descriptor that has a read pending on it is passed to its corresponding stream, request/response, or new connection handler. A stream handler will unserialize an input data item from the stream, perform computation on it yielding an output data item, which it passes to the mirror's data forwarder component.

The data forwarder will then serialize the item to all the open output streams. If a particular output stream is not writable, it will buffer the write and register a handler with the selector to be called when the stream is once again writable. This guarantees that the mirror's operation will not block due to an uncooperative communication target.

A request/response handler will unserialize the input data item from the file descriptor, perform computation yielding an output data item, and then serialize that output data item onto the same file descriptor. A new connection handler will simply accept the new connection, instantiate the appropriate handler for it (ie, stream or request response) and then register the handler with the connection manager.

The mirror class knows about a variety of different transport mechanisms, in particular, TCP, UDP (including multicast IP), Unix domain sockets, and pipes or file-like entities. The user asks the mirror to begin listening at a particular port for data or control messages either through an explicit mirror interface or by using EndPoints, which are objects that encapsulate all of a mirror's available transport mechanisms and can parse a string into an internal representation of a particular transport mechanism.

2.6.3 Example

Here is how the prediction server instantiates a prediction mirror that will receive measurements from a host named "pyramid" using TCP at port 5009, support reconfiguration requests via TCP at port 5010, and send predictions to all parties that connect via TCP at port 5011 or listen via multicast IP at address 239.99.99.99, port 5012, and also to standard out:

```
PredictionMirror mirror;
EndPoint tcpsource, tcpserver, tcpconnect
EndPoint multicasttarget, stdouttarget;

tcpsource.Parse("source:tcp:pyramid:5009");
tcpserver.Parse("server:tcp:5010");
tcpconnect.Parse("connect:tcp:5011");
multicasttarget.Parse("target:udp:239.99.99.99:5012");
stdouttarget.Parse("target:stdio:stdout");

mirror.AddEndPoint(tcpsource);
mirror.AddEndPoint(tcpserver);
mirror.AddEndPoint(tcpconnect);
mirror.AddEndPoint(multicasttarget);
mirror.AddEndPoint(stdouttarget);

mirror.Run();
```

In order to simplify writing clients for mirrors, we also implemented 3 reference classes, one for streaming input, one for streaming output, and one for request/response transactions. Here is how a client would begin to receive the multicasted prediction stream produced by the mirror code above:

```
EndPoint source;
StreamingInputReference<PredictionResponse> ref;
PredictionResponse pred;

source.Parse("source:udp:239.99.99.99:5012");
ref.ConnectTo(source);
while (...) {
    ref.GetNextItem(pred);
    pred.Print();
}
ref.Disconnect();
```

Similarly, here is the code that a client might use to reconfigure the prediction mirror via its TCP request/response interface, assuming the mirror is running on mojave:

```
EndPoint source;
Reference<PredictionReconfigurationRequest,
          PredictionReconfigurationResponse> ref;
PredictionReconfigurationRequest   req(...);
PredictionReconfigurationResponse resp;

source.Parse("source:tcp:mojave:5010");
ref.ConnectTo(source);
ref.Call(req,resp);
resp.Print();
```

2.7 Prediction components

Using the functionality implemented in the sensor libraries of Section 2.4, the time series prediction library of Section 2.5, and the mirror communication template library of Section 2.6, we implemented a set of *prediction components*. Each component is a program that implements a specific RPS function. On-line resource prediction systems are implemented by composing these components. The communication connectivity of a component is specified via command-line arguments, which means the location of the components and what transport any two components use to communicate can be determined at startup time. In addition, the components also support transient connections to allow run-time reconfiguration and to permit multiple applications to use their services. In Section 2.8.1 we compose an on-line host load prediction system out of the components we describe in this section.

We implemented a large set of prediction components, which are shown in Table 2.2. They fit into five basic groups: host load measurement, flow bandwidth measurement, measurement management, stream-based prediction, and request/response prediction.

The host load measurement and flow bandwidth measurement groups implement sensors and tools for working with them. In each group, the sensor component (eg, loadserver, flowbwserver) generates a stream of sensor-specific measurements, while the other components provide mechanisms to control the sensor, read the measurement streams, buffer the measurement streams to provide asynchronous request/response access to the measurements, and, finally, to convert sensor-specific measurements into a generic measurement type. The remainder of the components use these generic measurement streams.

The measurement management group provides tools for receiving generic measurement streams, buffering generic measurements, and accessing such buffers.

Component	Function
Host Load Measurement	
loadserver	Generates stream of host load measurements
loadclient	Prints loadserver's stream
loadreconfig	Changes a loadserver's host load measurement rate
loadbuffer	Buffers measurements with request/response access
loadbufferclient	Provides access to a loadbuffer
load2measure	Converts a load measurement stream to generic measurement stream
Flow Bandwidth Measurement	
flowbwserver	Generates stream of flow bandwidth measurements using Remos
flowbwclient	Prints flowbwserver's stream
flowbwreconfig	Reconfigures a running flowbwserver
flowbwbuffer	Buffers a flow bandwidth measurement stream with request/response access
flowbwbufferclient	Provides access to a flowbwbuffer
flowbw2measure	Converts a flow bandwidth measurement stream to generic measurement stream
Measurement Management	
measureclient	Prints a generic measurement stream
measurebuffer	Buffers generic measurements with request/response access
measurebufferclient	Provides access to a measurebuffer
Stream-based Prediction	
predserver	Computes predictions for a generic measurement stream
predserver_core	Performs actual computations to contain failures
predreconfig	Reconfigures a running predserver
evalfit	Evaluates a running predserver and reconfigures it when necessary
predclient	Prints a prediction stream
predbuffer	Buffers a prediction stream with request/response access
predbufferclient	Provides access to a predbuffer
Request/Response Prediction	
pred_reqresp_server	Computes "one-off" predictions for request/response clients
pred_reqresp_client	Makes "one-off" prediction requests on a pred_reqresp_server

Table 2.2: Prediction components implemented using RPS libraries.

The stream-oriented prediction group provides continuous prediction services for generic measurement streams. Predserver is the main component in this group. When started up, it retrieves a measurement sequence from a measurebuffer, fits the desired model to it, and then creates a predictor. As new measurements arrive in the stream, they are passed through the predictor to form m-step-ahead predictions and corresponding estimates of prediction error. These operations are similar to those described in Section 2.5.3. The actual work is done by a subprocess, predserver_core. This limits the impact of a crash caused by a bad model fit. If predserver_core crashes, predserver simply starts a new copy.

Predserver also provides a request/response control interface for changing the type of model, the length of the sequence to which the model is fit, and the number, m, of predictions it will make. This interface can be used by the user through the predreconfig program. Alternatively, and even at the same time, evalfit can use the interface. Evalfit receives a generic measurement stream and a prediction stream, and continuously evaluates the quality of the predictions using an evaluator as discussed in Section 2.5.2. When the prediction quality exceeds limits set by the user, evalfit will force the predserver it is monitoring to refit the model.

The remaining components in the stream-oriented prediction services group simply provide buffering and client functionality for prediction streams.

The request/response prediction group provides classic client/server access to the time series prediction library. Pred_reqresp_client sends a measurement sequence and a model template to pred_reqresp_server,

which fits a model and return predictions for the next m values of the sequence.

It is important to note that the set of of prediction components is not fixed. It is quite easy to construct new components using the libraries we described earlier. Indeed, we constructed additional components for the performance evaluation we describe in the next section.

2.8 Performance

The RPS-based prediction components described in the previous section are composed at startup time to form on-line prediction systems. To evaluate the performance of RPS for constructing such systems, we measured the RPS-based host load prediction system that is used in Chapters 5 and 6. We measured this representative system's performance in terms of the timeliness of its predictions, the maximum measurement rates that can be achieved, and the additional computational and communication load it places on the distributed system. In addition to the composed system, we also constructed a monolithic system using the RPS libraries directly and measured the maximum measurement rates it could support.

The conclusion of our study is that, for interesting measurement rates, both the composed and the monolithic systems can provide timely predictions using only tiny amounts of CPU time and network bandwidth. In addition, the maximum achievable measurement rates are 2 to 3 orders of magnitude higher than we currently need.

It is important to note that RPS is a toolkit for resource prediction, and, because of the inherent flexibility of such a design, it is difficult to measure RPS's performance for creating on-line resource prediction systems in a vacuum. Prediction components can be composed in many different ways to construct on-line resource prediction systems and the RPS libraries enable the construction of additional components or increased integration of functionality. Furthermore, prediction components can communicate in different ways. Finally, different resources require different measurement rates and predictive models. More complex predictive models require more computational resources while higher measurement rates require more computational and communication resources.

Because of the intractability of attempting to characterize this space, we instead focused on measuring the performance of the RPS-based on-line host load prediction system used in this thesis. The system is representative of RPS-based systems in the sense that it implements the functionality of Figure 2.1 using the prediction components. Furthermore, it is also a realistic system. It uses a predictive model that we have found appropriate for host load prediction, as discussed in Chapter 4. Finally, it is a fairly widely used system which has been distributed with Remos [82], and is currently used in QuO [145].

2.8.1 Host load prediction system

Figure 2.7 shows the configuration of prediction components used for host load prediction. The boxes in the figure represent prediction components while dark arrows represent stream communication between components, and symmetric arrows represent request/response communication between components. The arrows are annotated with communication volumes per cycle of operation of the system for streams and per call for request/response communication. s is the number of measurements being requested asynchronously from the measurebuffer while m is the number of steps ahead for which predictions are made and w is the number of predictions being requested from the predbuffer. Notice the similarity of this system to the high-level view of Figure 2.1.

The system works as follows. The loadserver component periodically measures the load on the host on which it is running using the GetLoadAvg library described in Section 2.4. Each new measurement is forwarded to any attached loadclients and also to load2measure, which converts it to a generic measurement
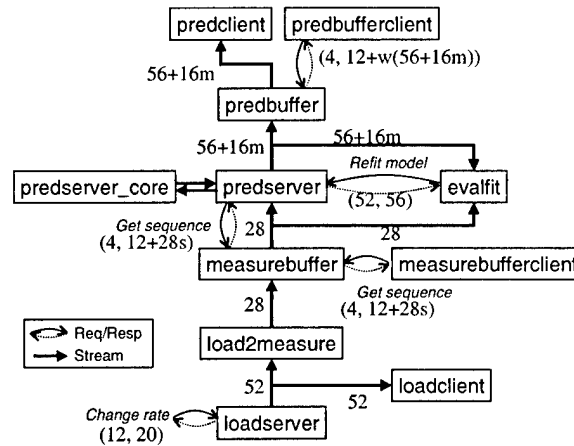
Figure 2.7: Online host load prediction system composed out of the RPS prediction components described in Section 2.7.

form and forwards it to measurebuffer. Measurebuffer buffers the last N measurements and provides request/response access to them. It also forwards the current measurement to predserver and evalfit. Predserver consumes the measurement and produces an m-step-ahead prediction using its subprocess, predserver_core. It forwards the prediction to predbuffer and to evalfit. Evalfit continuously compares predserver's predictions with the measurements it receives from measurebuffer and computes its own assessment of the quality of the predictions. For each new measurement, it compares its assessment with the requirements the user has specified as well as with the predictor's own estimates of their quality. When quality limits are exceeded it calls predserver to refit the model. Predserver's predictions also flow to predbuffer, which provides request/response access to some number of previous predictions and also forwards the predictions to any attached predclients. Predbufferclients can asynchronously request predictions from predbuffer. Of course, applications can decide, at any time, to access the prediction stream or the buffered predictions in the manner of predclient and predbufferclient.

Each measurement that loadserver produces is timestamped. This timestamp is passed along as the measurement makes its way through the system and is joined with a timestamp for when the corresponding prediction is completed, and for when the prediction finally arrives at an attached predclient. We shall use these timestamps to measure the latency from when a measurement is made to when its corresponding prediction is available for applications.

The system can be controlled in various ways. For example, the user can change loadserver's measurement rate, the predictive model that predserver uses, and the time horizon for predictions. We used the control over loadserver's measurement rate to help determine the computational and communication resources the system uses.

So far, we have not specified where each of the components runs or how the components communicate. As we discussed in the previous section, RPS lets us defer these decisions until startup time and even runtime. In the study we describe in this section, we ran all of the components on the same machine and arrange for them to communicate using TCP. The machine we used is a 500 MHz Alpha 21164-based DEC personal workstation.

This configuration of prediction components is an interesting one to measure. It is reasonable to run all the components on a single machine since relatively low measurement rates and reasonably simple predictive models are sufficient for host load prediction, as we will show in Chapters 3 and 4, respectively. It would be more efficient to use a local IPC mechanism such as pipes or Unix domain sockets to communicate between components. Indeed, a production host load prediction system might very well be implemented

as a single process. We briefly discuss the performance of such an implementation in Section 2.8.3. TCP is interesting to look at because it gives us some idea of how well an RPS-based system might perform running on multiple hosts, which might be desirable, for, say, network bandwidth prediction. Furthermore, if RPS can achieve reasonable performance levels in such a flexible configuration, it is surely the case that a performance-optimized RPS-based system would do at least as well.

The predictive model that is used is an AR(16) fit to 600 samples and evalfit is configured so that model refitting does not occur. Predictions are made 30 steps into the future. The default measurement rate is 1 Hz. This model and rate is appropriate for host load prediction, as we discuss in Chapters 3 and 4.

The following illustrates how the various prediction components are started:

```
% loadserver 1000000 server:tcp:5000 connect:tcp:5001 &
% loadclient source:tcp:'hostname':5001 &
% load2measure 0 source:tcp:'hostname':5001 connect:tcp:5002 &
% measurebuffer 1000 source:tcp:'hostname':5002
      server:tcp:5003 connect:tcp:5004 &
% predserver source:tcp:'hostname':5004
      source:tcp:'hostname':5003 server:tcp:5005 connect:tcp:5006 &
% evalfit source:tcp:'hostname':5004 source:tcp:'hostname':5006
      source:tcp:'hostname':5005
      30 999999999 1000.0 999999999 600 30 AR 16  &
% predbuffer 100 source:tcp:'hostname':5006 server:tcp:5007
      connect:tcp:5008 &
% predclient source:tcp:'hostname':5008  &
```

The use of measurebufferclient and predbufferclient are not shown above since these are run only intermittently.

2.8.2 Limits

Before we present the details of the performance of the host load prediction system, it is a good idea to understand the limits of achievable performance on this machine. Recall from Section 2.4 that the host load sensor library requires only about 1.6 μs to acquire a sample. As for the cost of prediction, Figure 2.4 indicates that fitting and initializing an AR(16) model on 600 data points requires about 1 ms of CPU time, with a step/predict time of about 100 μs. The computation involved in evalfit, load2measure, and the various buffers amounts to about 50 μs, thus the total computation time per cycle is 151.6 μs. If no communication was involved, we would expect the prediction system to operate at a rate no higher than 6.6 KHz.

However, the prediction system also performs communication. Examination of Figure 2.7 indicates that, for 30-step-ahead ($m = 30$) predictions, eight messages are sent for each cycle. There are 3 28 byte messages, 2 52 byte messages, and 3 536 byte messages. The measured bandwidths of the host for messages of this size are 2.4 MB/s (28 bytes), 4.2 MB/s (52 bytes), and 15.1 MB/s (536 bytes). Therefore the lower bound transfer times for these messages are 11.7 μs (28 bytes), 12.4 μs (52 bytes), and 35.5 μs (536 bytes). The total communication time per cycle is therefore at least $(3)11.7 + (2)12.4 + (3)35.5 = 166.4$ μs, and the total time per cycle is at least $151.6 + 166.4 = 318$ μs, which suggests a corresponding upper bound on system's rate of about 3.1 KHz.

It is important to note that these rates are far in excess of the 1 Hz rate we expect from a host load prediction system, or even the 14 Hz peak rate for our Remos-based network sensor. What these high rates suggest, however, is that, for rates of interest to us, we can expect the prediction system to use only a tiny percentage of the CPU. In terms of communication load, we will only place $(3)28 + (2)52 + (3)536 = 1796$ bytes onto the network per cycle. At a rate of 14 Hz, this amounts to about 25 KB/s of traffic.
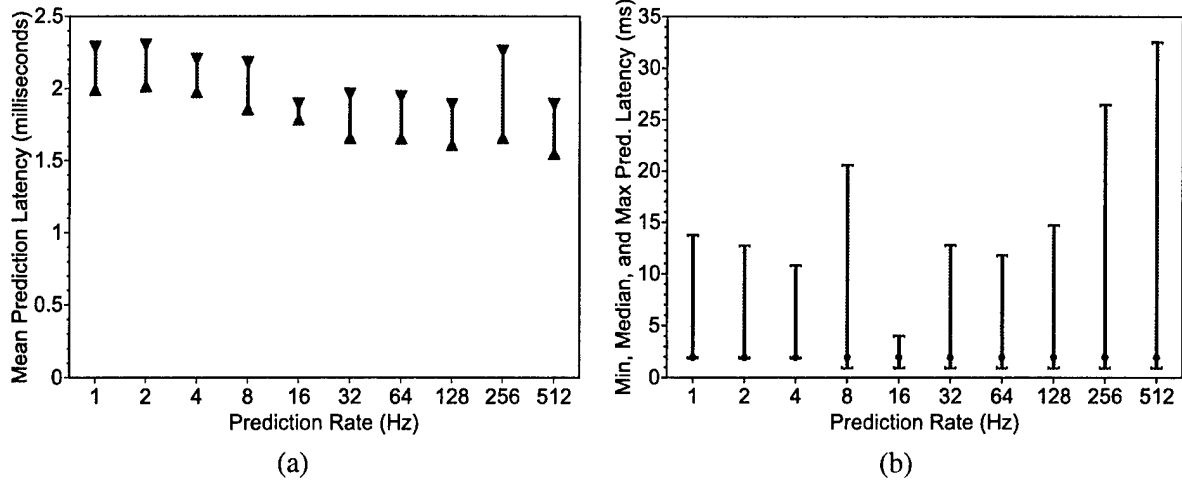
Figure 2.8: Prediction latency as a function of measurement rate: (a) 95% confidence interval of mean latency, (b) Minimum, median, maximum latency

Of course, these are the upper limits of what is possible. We would expect that overheads of the mirror communication template library and the data copying implied by the TCP communication we use to result in lower performance levels.

The host we evaluated the system on has a timer interrupt rate of 1024 Hz, which means the all measurement rates in excess of this amount to "as fast as possible." This rate also results in a clock accuracy of approximately one millisecond.

2.8.3   Evaluation

We configured the host load prediction system so that the model will be fit only once, and thus measured the system in steady state. We measured the prediction latency, communication bandwidth, and the CPU load as functions of the measurement rate, which we swept from 1 Hz to 1024 Hz in powers of 2. We found that the host load prediction system can sustain measurement rates of 730 Hz with mean and median prediction latencies of around 2 ms. For measurement rates that are of interest to us, such as the 1 Hz rate for load and the 14 Hz for flow bandwidth, the additional load the system places on the machine is minimal.

Prediction latency

In an on-line prediction system, the timeliness of the predictions is paramount. No matter how good a prediction is, it is useless if if it does not arrive sufficiently earlier than the measurement it predicts. We measured this timeliness in the host load prediction system as the latency from when a measurement becomes available to when the prediction it generates becomes available to applications that are interested in it. This is the latency from the loadserver component to the predclient component in Figure 2.7.

The prediction latency should be independent of the measurement rate until the prediction system's computational or communication resource demands saturate the CPU or the network. Figure 2.8 shows that this is indeed the case. Figure 2.8(a) plots the 95% confidence interval for the mean prediction latency as a function of increasing measurement rates. We do not plot the latency for the 1024 Hz rate since at this point the CPU is saturated and the latency increases with backlogged predictions. Up to this point, the mean prediction latency is roughly 2 ms.

Figure 2.8(b) plots the minimum, median, and maximum prediction latencies as a function of increasing
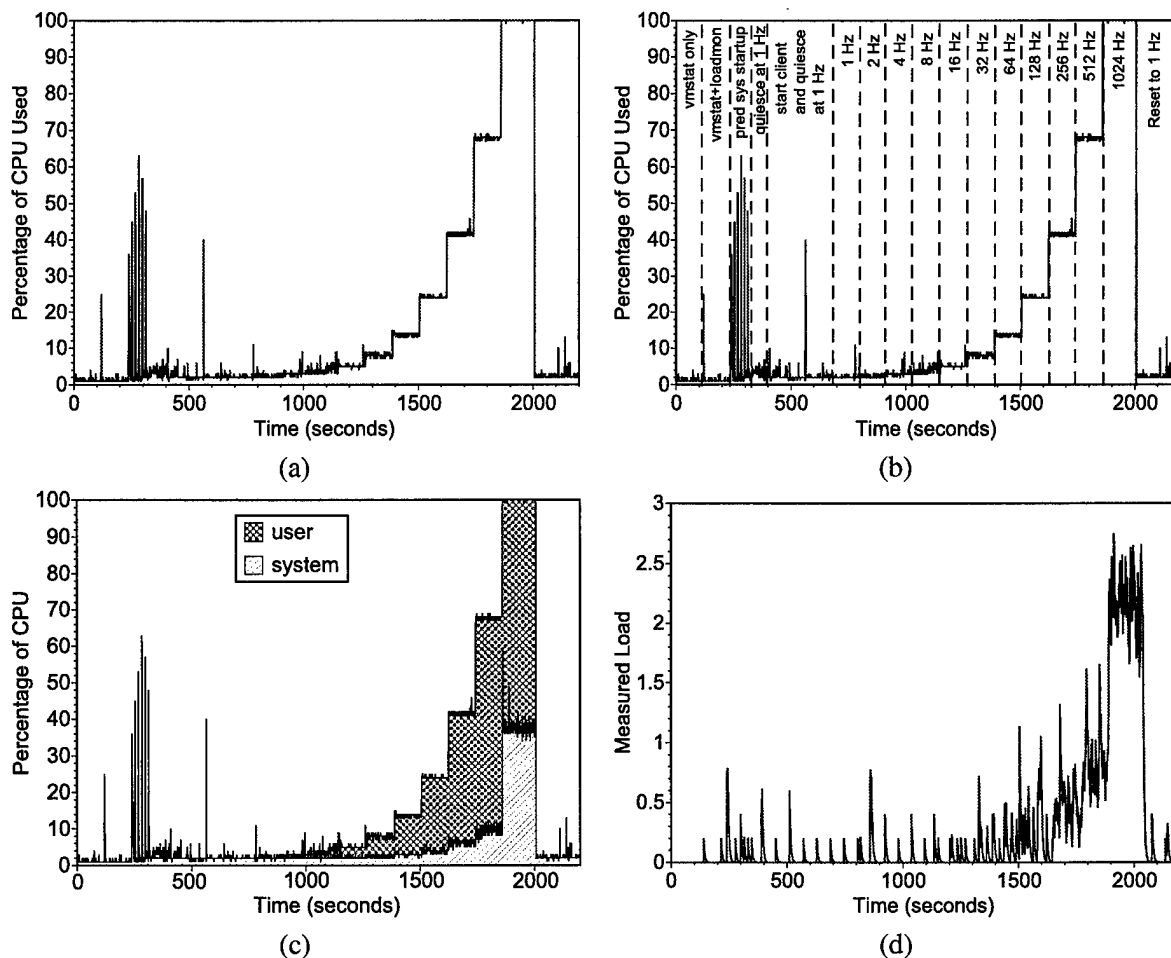
Figure 2.9: CPU load produced by system. The measurement rate is swept from 1 Hz to 1024 Hz. (a) shows total percentage of CPU used over time, (b) is the same as (a) but includes operational details, (c) shows user and system time, (d) shows load average.

measurement rate. Once again, we have elided the 1024 Hz rate since latency begins to grow with backlog. The median latency is 2 ms, while the minimum latency is at 1 ms, which is the resolution of the timer we used. The highest latency we saw was 33 ms.

Resource usage

In addition to providing timely predictions, an on-line resource prediction system should also make minimal resource demands. After all, the purpose of the system is to predict resource availability for applications, not to consume the resources for itself.

To measure the CPU usage of our representative host load prediction system, we did the following. First, we started two resource monitors, vmstat and our own host load sensor. Vmstat is run every second and prints the percentage of the last second that has been charged to system and user time. Our host load sensor measures the average run queue length every second. After the sensors were started, we started the prediction system at its default rate of 1 Hz and let it quiesce. Next, we started a predclient and let the system quiesce. Then, we swept the measurement rate from 1 Hz to 1024 Hz in powers of 2. For each of the 10 rates, we let the system quiesce. Finally, we reset the rate to 1 Hz. Figure 2.9 shows plots of what the

Rate (Hz)	Bytes/sec
1	1796
2	3592
4	7184
8	14368
16	28736
32	57472
64	114944
128	229888
256	459776
512	919552
1024	1839104

Table 2.3: Bandwidth requirements as a function of measurement rate.

sensors recorded over time.

Figure 2.9(a) shows the percentage of the CPU that was in use over time, as measured by vmstat. Figure 2.9(b) is the same graph, annotated with the operational details described above. Figure 2.9(c) breaks down the CPU usage into its system and user components. The system component is essentially the time spent doing TCP-based IPC between the different components. Figure 2.9(d) shows the output of the load average sensor. When the load measured by this sensor exceeds one, we have saturated the CPU.

There are several important things to notice about Figure 2.9. First, we can sustain a measurement rate of between 512 Hz and 1024 Hz on this machine. Interpolating, it seems that we can sustain about a 730 Hz rate using TCP-based IPC, or about 850 Hz ignoring the system-side cost of IPC. While this is nowhere near the upper bound of 3.1 KHz that we arrived at in Section 2.8.2 , it is still much faster than we actually need for the purposes of host load prediction (1 Hz) and than the limits of our network flow bandwidth sensor (14 Hz).

In Section 2.8.3, we compare the maximum rate achievable by this composed host load prediction system to a monolithic system. The monolithic system achieves much higher rates overall, and those rates are closer to the upper bound.

A second observation is that for these interesting 1 and 14 Hz rates, CPU usage is quite low. At 1 Hz, it is around 2% while at 16 Hz (closest rate to 14 Hz) it is about 5%. For comparison, the "background" CPU usage measured when only running the vmstat probe is itself around 1.5%. Figure 2.9(d) shows that this is also the case when measured by load average.

Table 2.3 shows the bandwidth requirements of the system at the different measurement rates. To understand how small these requirements are, consider a 1 Hz host load prediction system running on each host in the network and multicasting its predictions to each of the other hosts. Approximately 583 hosts could multicast their prediction streams in 1 MB/s of sustained traffic, with each host using only 0.5% of its CPU to run its prediction system. Alternatively, 42 network flows measured at the maximum rate could be predicted. If each host or flow only used the network to provided asynchronous request/response access to its predictions, many more hosts and flows could be predicted. For example, if prediction requests from applications arrived at a rate of one per host per second, introducing 552 bytes of traffic per prediction request/response transaction, 1900 hosts could operate in 1 MB/s.

A monolithic system

The composed host load prediction system we have described so far can operate at a rate 52–730 times higher than we need and uses negligible CPU and communication resources at the rates at which we actually desire to operate it. However, the maximum rate it can sustain is only 24% of the upper bound we determined in

System	Transport	Optimal Rate	Measured Rate	Percent of Optimal
Monolithic	In-process	6.6 KHz	5.3 KHz	80 %
Monolithic	Unix domain socket	5.5 KHz	3.6 KHz	65 %
Monolithic	TCP	5.3 KHz	2.7 KHz	51 %
Composed	TCP	3.1 KHz	720 Hz	24 %

Table 2.4: Maximum measurement rates achieved by monolithic and composed host load prediction systems.

Section 2.8.2. To determine if higher rates are indeed possible, we implemented a monolithic, single process host load prediction system using the RPS libraries directly. This design can sustain a peak rate of 2.7 KHz when configured to use TCP, which is almost four times higher than the composed system.

Table 2.4 shows the maximum rates the monolithic system achieved for three transports: in process, where the client is in the same process; Unix domain socket, where the (local) client listens to the prediction stream through a Unix domain socket; and TCP, where the client operates as with the earlier system. For comparison, it also includes the maximum rate of the composed system described earlier. In each case, we also show the optimal rate, which is derived in a manner similar to Section 2.8.2. The in-process case shows us the overhead of using the mirror communication template library, which enables considerable flexibility. That overhead is approximately 20%. The domain socket and TCP cases include additional, unmodeled overheads that are specific to these transports.

2.9 Conclusion

We have developed a resource signal methodology which researchers can use to attack resource availability prediction problems using signal analysis and prediction techniques. Finding a scarcity of tools to carry out the latter steps of the methodology, we designed, implemented, and evaluated RPS, an extensible toolkit for constructing on-line and off-line resource prediction systems in which resources are represented by independent, periodically sampled, scalar-valued measurement streams. RPS consists of resource sensor libraries, an extensive time series prediction library, a sophisticated communication library, and a set of prediction components out of which resource prediction systems can be readily composed. The performance of RPS is quite good. The host load prediction system that we advocate later in this dissertation provides timely predictions with minimal CPU and network load at reasonable measurement rates. These results support the feasibility of resource prediction in general, and of using RPS-based systems for resource prediction in particular.

In the next two chapters, we put the resource signal methodology and RPS to work to understand and predict host load. The RPS-based host load prediction system we develop is then used as the basis of the running time and real-time scheduling advisors of Chapters 5 and 6.

Chapter 3

Statistical Properties of Host Load

This dissertation argues for basing real-time scheduling advisors on explicit resource-oriented prediction, specifically on the prediction of resource signals. In the last chapter, we presented a methodology for understanding and predicting such resource signals and described our toolkit for carrying out the methodology. This chapter applies the first four steps of our resource signal methodology, as described in Section 2.1, to *host load*, a signal that tracks CPU availability. The essence of these steps is to choose an appropriate resource signal, determine how to sample it, collect representative traces of the signal to form a signal analysis and prediction problem, and then analyze the traces to find appropriate predictive models. The next chapter carries out the final two steps of the methodology, resulting in an appropriate predictive model for host load, and an on-line host load prediction system.

We ultimately want to predict the running time of tasks, which varies as a result of changing CPU availability. CPU availability is well measured by the host load signal we study in this chapter. The running time of a compute-bound task is directly related to the average load it encounters during execution. What, then, are the qualitative and quantitative properties of load on real systems, and what are the implications of these properties for host load prediction? Since we are interested in scheduling short tasks as well as long ones, the answers to these questions should extend to correspondingly short timescales. Unfortunately, to date there has been little work on characterizing the properties of load at fine resolutions. The available studies concentrate on understanding functions of load, such as availability [92] or job durations [40, 77, 60]. Furthermore, they deal with the coarse grain behavior of load—how it changes over minutes, hours and days.

To understand the properties of host load on real systems at fine resolutions, we collected week-long, 1 Hz resolution traces of the Digital Unix load average (specifically, an exponential average with a five second time constant) on over 35 different machines that we classify as production and research cluster machines, compute servers, or desktop workstations. We collected two sets of such traces at different times of the year. The 1 Hz sample rate is sufficient to capture all of the dynamic load information that is available to user-level programs running on these machines. This chapter presents a detailed statistical analysis of both sets of traces and their implications for prediction. The subsequent chapters use these traces to evaluate the host load prediction system, the running time advisor, and the real-time scheduling advisor.

The basic question is whether load traces that might seem at first glance to be random and unpredictable might have structure that could be exploited for prediction. Our results suggest that host load signals do indeed have some structure in the form of clearly identifiable properties. In essence, our results characterize how load varies, which should be of interest not only to developers of prediction algorithms, but also to those who need to generate realistic synthetic loads in simulators or to those doing analytic work. The traces can also be used to reconstruct real background workloads using host load trace playback, as described in Chapter 5.

We found that host load is a resource signal with high absolute and relative variability, leading to widely
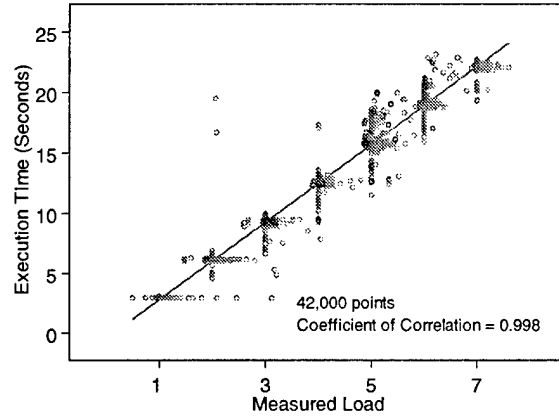
Figure 3.1: Relationship between average load during execution and running time

varying running times. While this is distressing, host load signals are not random. Time series analysis shows that there is strong correlation over time. This suggests that linear time series models, which attempt to capture this autocorrelation parsimoniously, may be appropriate for predicting host load. However, host load signals are also self-similar, which suggests that complex predictive models, such as the ARFIMA models described in the previous chapter, may be necessary to predict them. Finally, host load exhibits what we term epochal behavior—the signal remains stationary for an extended period of time, and then abruptly transitions to another stationary regime. Linear time series models, even those that explicitly model nonstationarity, cannot model such abrupt transitions. The implication is that a linear model would have to be refit whenever such a transition occurred. Such transitions might be explicitly detected using changepoint detection techniques or implicitly detected by suddenly large prediction errors. The latter is the method used by the system described in this dissertation.

3.1 Host load and running time

We chose to study the host load signal because, for compute-bound tasks, there is an intuitive relationship between host load and running time. Consider Figure 3.1, which plots running time versus average load experienced during execution for tasks consisting of simple wait loops. The data was generated by running variable numbers of these tasks together, at identical priority levels, on an otherwise unloaded Digital Unix machine. Each of these tasks sampled the Digital Unix five-second load average at roughly one second intervals during their execution and at termination printed the average of these samples as well as their running time. It is these pairs that make up the 42,000 points in the figure. Notice that the relationship between the measured load during execution and the running time is almost perfectly linear ($R^2 > 0.99$).

If load were presented as a continuous signal, we would summarize this relationship between running time and load as

$$\frac{t_{exec}}{1 + \frac{1}{t_{exec}} \int_0^{t_{exec}} z(t)dt} = t_{nom} \tag{3.1}$$

where t_{nom} is the running time of the task on a completely unloaded machine, $1 + \frac{1}{t_{exec}} \int_0^{t_{exec}} z(t)dt$ is the average load experienced during execution ($z(t)$ is the continuous "background" load), and t_{exec} is the task's running time. In practice, we can only sample the load with some noninfinitesimal sample period Δ so we can only approximate the integral by summing over the values in the sample sequence. For these machines, $\Delta = 1$ second is appropriate, as we show in the next section. We will write a sequence of load samples,

which we will also refer to as a *load signal*, as $\langle z_t \rangle = \ldots, z_{t-1}, z_t, z_{t+1}, \ldots$

3.2 Measurement methodology

Having decided that host load is a useful signal, we next developed a sensor for it, and determined how quickly to sample it. The load on a Unix system at any given instant is the number of processes that are running or are ready to run, which is the length of the ready queue maintained by the scheduler. The kernel samples the length of the the ready queue at some rate and exponentially averages some number of previous samples to produce a load average which can be accessed from a user program. The specific Unix system we used was Digital Unix (DUX).

Unlike many Unix implementations, which exponentially average with a time constant of one minute at the finest, DUX uses a time constant of five seconds. This small time constant allows us to capture considerably more of the dynamics of load than would have been possible on other Unix implementations, and it minimizes the effect of phantom correlations due to the exponential filter. Interestingly, directly sampling the length of the ready queue, which we tried on Windows NT, does not provide much useful information because it is impossible to sample the queue fast enough from a user process.

We developed a small tool to sample the DUX load average at one second intervals and log the resulting time series to a data file. The tool is based on the host load sensor described in the previous chapter. The 1 Hz sample rate was arrived at by subjecting DUX systems to varying loads and sampling at progressively higher rates to determine the rate at which DUX actually updated the value. DUX updates the value at a rate of $1/2$ Hz, thus we chose a 1 Hz sample rate by the Nyquist criterion. This choice of sample rate means we capture all of the dynamic load information the operating system makes available to user programs.

3.3 Description of traces

Having chosen an appropriate resource signal and sampling rate, we next carried out the second step of the resource signal methodology: collecting traces. We ran our trace collection tool on 39 hosts belonging to the Computing, Media, and Communication Laboratory (CMCL) at CMU and the Pittsburgh Supercomputing Center (PSC) for slightly more than one week in late August, 1997. A second set of week-long traces was acquired on almost exactly the same set of machines (35 machines total) in late February and early March, 1998. The results of the statistical analysis were similar for the two sets of traces.

All of the hosts in the August, 1997 set were DEC Alpha DUX machines, running either DUX 3.2 or 4.0 and they form four classes:

- *Production Cluster*: 13 hosts of the PSC's "Supercluster", including two front-end machines (axpfea, axpfeb), four interactive machines (axp0 through axp3), and seven batch machines scheduled by a DQS [69] variant (axp4 through axp10).

- *Research Cluster*: eight machines in an experimental cluster in the CMCL (manchester-1 through manchester-8).

- *Compute servers*: two high performance large memory machines used by the CMCL group as compute servers for simulations and the like (mojave and sahara).

- *Desktops*: 16 desktop workstations owned by members of the CMCL (aphrodite through zeno).

The same hosts were used for the March, 1998 traces, with the following exceptions:

- *Production Cluster*: axp9 was replaced by axp11 due to hardware failures.

Hostname	Start Time	Days	Samples
Production Cluster			
axp0.psc	Tue Aug 12 21:29:12 EDT 1997	15.00	1296000
axp1.psc	Tue Aug 12 21:30:09 EDT 1997	14.00	1209600
axp2.psc	Tue Aug 12 21:30:53 EDT 1997	14.00	1209600
axp3.psc	Tue Aug 12 21:31:13 EDT 1997	14.00	1209600
axp4.psc	Tue Aug 12 21:31:12 EDT 1997	14.00	1209600
axp5.psc	Tue Aug 12 21:31:47 EDT 1997	14.00	1209600
axp6.psc	Tue Aug 12 21:31:15 EDT 1997	15.00	1296000
axp7.psc	Tue Aug 12 20:51:19 EDT 1997	13.00	1123200
axp8.psc	Tue Aug 12 21:31:19 EDT 1997	14.00	1209600
axp9.psc	Tue Aug 12 21:31:45 EDT 1997	14.00	1209600
axp10.psc	Tue Aug 12 21:31:21 EDT 1997	14.00	1209600
axpfea.psc	Sat Aug 16 14:44:29 EDT 1997	13.00	1123200
axpfeb.psc	Sat Aug 16 14:44:55 EDT 1997	12.00	1036800
Research Cluster			
manchester-1.cmcl	Sun Aug 17 19:41:10 EDT 1997	3.92	338400
manchester-2.cmcl	Sun Aug 17 19:41:09 EDT 1997	4.00	345600
manchester-3.cmcl	Sun Aug 17 19:41:13 EDT 1997	3.96	342000
manchester-4.cmcl	Sun Aug 17 19:41:10 EDT 1997	4.00	345600
manchester-5.cmcl	Sun Aug 17 19:41:09 EDT 1997	4.04	349200
manchester-6.cmcl	Sun Aug 17 19:41:09 EDT 1997	4.08	352800
manchester-7.cmcl	Sun Aug 17 19:41:10 EDT 1997	4.00	345600
manchester-8.cmcl	Sun Aug 17 19:41:10 EDT 1997	4.00	345600
Compute Servers			
mojave.cmcl	Sun Aug 17 19:41:11 EDT 1997	4.04	349200
sahara.cmcl	Sun Aug 17 19:41:11 EDT 1997	4.00	345600
Desktops			
aphrodite.nectar	Sun Aug 17 19:41:12 EDT 1997	4.00	345600
argus.nectar	Sun Aug 17 19:41:17 EDT 1997	4.04	349200
asbury-park.nectar	Sun Aug 17 19:41:11 EDT 1997	4.00	345600
asclepius.nectar	Sun Aug 17 19:41:07 EDT 1997	4.08	352800
bruce.nectar	Sun Aug 17 19:41:10 EDT 1997	3.92	338400
cobain.nectar	Sun Aug 17 19:41:12 EDT 1997	4.04	349200
darryl.nectar	Sun Aug 17 19:41:32 EDT 1997	1.71	147600
hawaii.cmcl	Sun Aug 17 19:41:11 EDT 1997	2.63	226800
hestia.nectar	Sun Aug 17 19:41:12 EDT 1997	4.00	345600
newark.cmcl	Sun Aug 17 19:41:13 EDT 1997	4.00	345600
pryor.nectar	Sun Aug 17 19:41:13 EDT 1997	1.71	147600
rhea.nectar	Sun Aug 17 19:41:11 EDT 1997	4.00	345600
rubix.mc	Sun Aug 17 19:41:13 EDT 1997	4.00	345600
themis.nectar	Sun Aug 17 19:41:09 EDT 1997	4.00	345600
uranus.nectar	Sun Aug 17 19:41:13 EDT 1997	4.00	345600
zeno.nectar	Sun Aug 17 19:41:13 EDT 1997	4.08	352800

Table 3.1: Details of the August, 1997 traces.

- *Desktops*: argus, asclepius, bruce, cobain, darryl, and hestia were replaced by belushi and loman due to hardware upgrades.

Tables 3.1 and 3.2 provide additional details of the individual August, 1997 and March, 1998 traces. The author will be happy to provide the traces to any interested readers.

Hostname	Start Time	Days	Samples
Production Cluster			
axp0.psc	Wed Feb 25 17:34:26 EST 1998	12.08	1043400
axp1.psc	Wed Feb 25 17:34:25 EST 1998	12.08	1043400
axp2.psc	Wed Feb 25 17:34:25 EST 1998	12.08	1043400
axp3.psc	Wed Feb 25 17:34:26 EST 1998	12.03	1039400
axp4.psc	Wed Feb 25 17:34:27 EST 1998	12.06	1041900
axp5.psc	Wed Feb 25 17:34:27 EST 1998	12.03	1039700
axp6.psc	Wed Feb 25 17:34:27 EST 1998	12.08	1043400
axp7.psc	Wed Feb 25 17:34:27 EST 1998	12.01	1037700
axp8.psc	Wed Feb 25 17:34:28 EST 1998	12.06	1041800
axp10.psc	Wed Feb 25 17:34:28 EST 1998	12.03	1039700
axp11.psc	Wed Feb 25 17:16:20 EST 1998	12.03	1039800
axpfea.psc	Wed Feb 25 17:18:01 EST 1998	12.08	1043800
axpfeb.psc	Wed Feb 25 17:23:34 EST 1998	12.0	1043400
Research Cluster			
manchester-1.cmcl	Wed Feb 25 20:42:30 EST 1998	8.36	721900
manchester-2.cmcl	Wed Feb 25 20:42:23 EST 1998	8.36	721900
manchester-3.cmcl	Wed Feb 25 20:42:23 EST 1998	8.36	721900
manchester-4.cmcl	Wed Feb 25 20:42:29 EST 1998	8.35	721300
manchester-5.cmcl	Wed Feb 25 20:42:27 EST 1998	8.36	721900
manchester-6.cmcl	Wed Feb 25 20:42:30 EST 1998	8.36	721900
manchester-7.cmcl	Wed Feb 25 20:42:24 EST 1998	8.35	721800
manchester-8.cmcl	Wed Feb 25 20:42:26 EST 1998	8.35	721800
Compute Servers			
mojave.cmcl	Wed Feb 25 20:42:31 EST 1998	5.31	458800
sahara.cmcl	Wed Feb 25 20:42:34 EST 1998	8.34	721300
Desktops			
aphrodite	Wed Feb 25 20:42:17 EST 1998	8.36	722000
asbury-park	Wed Feb 25 20:42:23 EST 1998	5.90	509400
belushi	Wed Feb 25 20:42:28 EST 1998	7.77	671400
hawaii	Wed Feb 25 20:42:18 EST 1998	8.36	722000
loman	Wed Feb 25 20:42:34 EST 1998	8.36	721900
newark.cmcl	Wed Feb 25 20:42:29 EST 1998	8.36	722200
pryor.nectar	Wed Feb 25 20:42:24 EST 1998	8.36	722100
rhea.nectar	Wed Feb 25 21:12:17 EST 1998	10.19	880600
rubix.mc	Wed Feb 25 20:42:24 EST 1998	1.81	156400
themis.nectar	Wed Feb 25 20:42:29 EST 1998	4.94	426900
uranus.nectar	Wed Feb 25 20:42:33 EST 1998	8.36	722000
zeno.nectar	Wed Feb 25 20:42:26 EST 1998	6.23	537900

Table 3.2: Details of the March, 1998 traces.

3.4 Statistical analysis

We analyzed the individual load traces using summary statistics, histograms, fitting of analytic distributions, and time series analysis. The picture that emerges is that load varies over a wide range in very complex ways. Load distributions are rough and frequently multi-modal. Even traces with unimodal histograms are not well fitted by common analytic distributions, which have tails that are either too short or too long. Time series analysis shows that load is strongly correlated over time, but also has complex, almost noise-like frequency domain behavior.

We summarized each of our load traces in terms of our statistical measures and computed their corre-

	Mean Load	Sdev Load	COV Load	Max Load	Max/Mean Load	Mean Epoch	Sdev Epoch	COV Epoch	Hurst Param	Entropy
Mean Load	1.00									
Sdev Load	0.53	1.00								
COV Load	-0.49	-0.22	1.00							
Max Load	0.60	0.18	-0.32	1.00						
Max/Mean Load	-0.36	-0.39	0.51	0.03	1.00					
Mean Epoch	-0.04	-0.10	-0.19	0.08	-0.05	1.00				
Sdev Epoch	-0.02	-0.10	-0.20	0.09	-0.06	0.99	1.00			
COV Epoch	0.07	-0.11	-0.23	0.15	-0.02	0.95	0.96	1.00		
Hurst Param	0.45	0.58	-0.21	0.03	-0.49	0.08	0.10	0.18	1.00	
Entropy	0.42	0.51	-0.10	0.40	-0.36	-0.27	-0.25	-0.30	0.24	1.00

(a) Correlations for August, 1997 traces

	Mean Load	Sdev Load	COV Load	Max Load	Max/Mean Load	Mean Epoch	Sdev Epoch	COV Epoch	Hurst Param	Entropy
Mean Load	1.00									
Sdev Load	0.72	1.00								
COV Load	-0.64	-0.48	1.00							
Max Load	0.43	0.11	-0.25	1.00						
Max/Mean Load	-0.48	-0.49	0.93	-0.07	1.00					
Mean Epoch	-0.12	-0.23	-0.08	0.17	-0.05	1.00				
Sdev Epoch	-0.13	-0.22	-0.09	0.15	-0.06	0.99	1.00			
COV Epoch	-0.03	-0.12	-0.15	0.23	-0.11	0.88	0.93	1.00		
Hurst Param	-0.30	-0.41	0.29	0.15	0.36	0.92	0.90	0.78	1.00	
Entropy	0.05	0.27	-0.19	-0.05	-0.27	-0.19	-0.18	-0.17	-0.29	1.00

(b) Correlations for March, 1998 traces

Figure 3.2: Correlation coefficients (CCs) between all of the discussed statistical properties.

lations to determine how the measures are related. Figure 3.2(a) contains the correlations for the August, 1997 set while Figure 3.2(b) contains the correlations for the March, 1998 set. Unless otherwise noted, the remaining figures in the chapter similarly present independent results for the two sets of traces. Each cell of the tables in Figure 3.2 is the correlation coefficient (CC) between the row measure and the column measure, computed over the the load traces in the set. We will refer back to the highlighted cells (where the absolute correlations are greater than 0.3 [1]) throughout the chapter. It is important to note that these cross correlations can serve as a basis for clustering load traces into rough equivalence classes. Note that the correlations in several cases (eg, Hurst parameter vs. mean load) are significantly different between the two sets of traces. We shall point out and attempt to explain these differences when we encounter them.

3.4.1 Summary statistics

Summarizing each load trace in terms of its mean, standard deviation, and maximum and minimum illustrates the extent to which load varies. Figure 3.3 shows the mean load and the +/- one standard deviation points for each of the traces. As we might expect, the mean load on desktop machines is significantly lower than on other machines. However, we can also see a lack of uniformity within each class, despite the long duration of the traces. This is most clear among the Production Cluster machines, where fewer than half of the machines seem to be doing most of the work. This lack of uniformity even over long time scales shows clear opportunity for load balancing or resource management systems.

From Figure 3.3 we can also see that desktop machines have smaller standard deviations than the other machines. Indeed, the standard deviation, which shows how much load varies in *absolute* terms, grows with increasing mean load (Figure 3.2 shows CC=0.53 for the 1997 traces and CC=0.72 for the 1998 traces). However, in *relative* terms, variance shrinks with increasing mean load. This can be seen in Figure 3.4,

[1]This cutoff was chosen to select only those correlations different from zero with high significance.
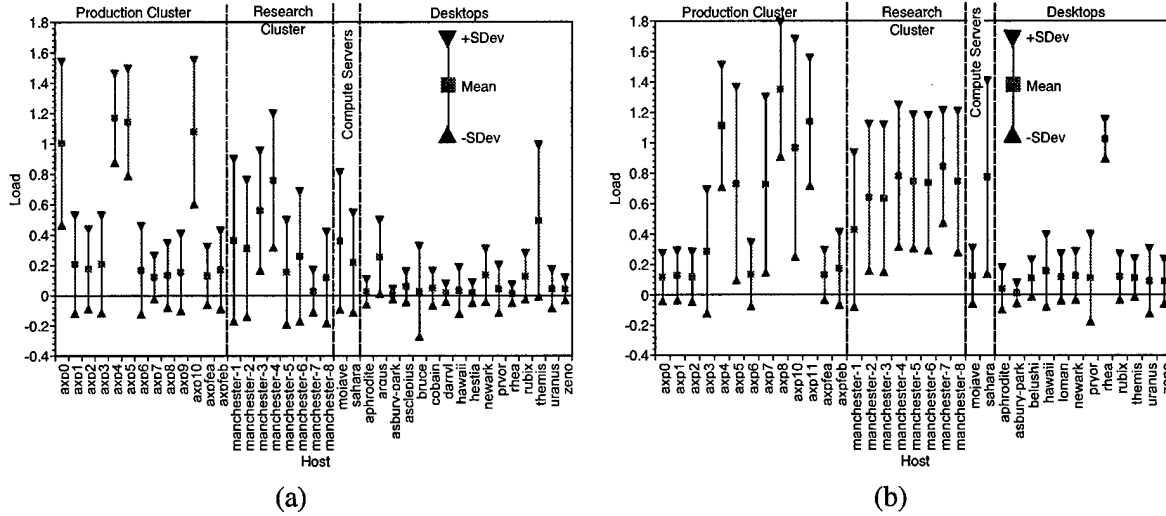
Figure 3.3: Mean load +/- one standard deviation: (a) August, 1997 traces, (b) March, 1998 traces.
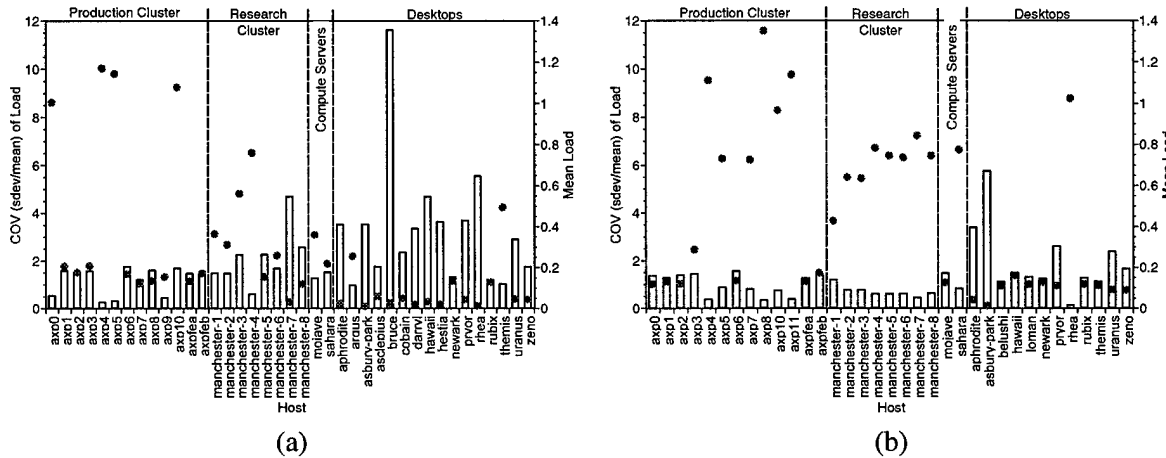
Figure 3.4: COV of load and mean load: (a) August, 1997 traces, (b) March, 1998 traces.

which plots the coefficient of variation (the standard deviation divided by the mean, abbreviated as the COV) and the mean load for each of the load traces. Here we can see that desktop machines, with their smaller mean loads, have large COVs compared to the other classes of machines. The CC between mean load and the COV of load is -0.49 for the 1997 traces and -0.64 for the 1998 traces. It is clear that as load increases, it varies *less* in relative terms and *more* in absolute terms.

This difference between absolute and relative behavior also holds true for the maximum load. Figure 3.5 shows the minimum, maximum, and mean load for each of the traces. The minimum load is, not surprisingly, zero in almost every case. The maximum load is positively correlated with the mean load (CC=0.60 for the 1997 traces and CC=0.43 for the 1998 traces in Figure 3.2). Figure 3.6 plots the ratio max/mean and the mean load for each of the traces. It is clear that this relative measure is inversely related to mean load, and Figure 3.2 shows that the CC is -0.36 for the 1997 traces and -0.48 for the 1998 traces. It is also important to notice that while the differences in maximum load between the hosts are rather small (Figure 3.5), the differences in the max/mean ratio can be quite large (Figure 3.6). Desktops are more surprising machines in relative terms.

With respect to scheduling tasks, the implication of the differences between relative and absolute mea-
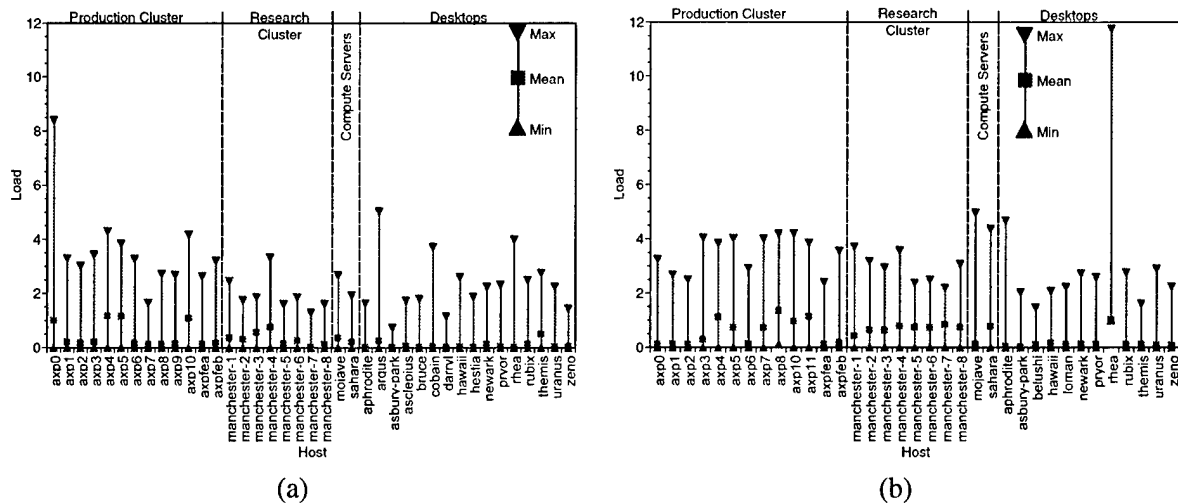
Figure 3.5: Minimum, maximum, and mean load: (a) August, 1997 traces, (b) March, 1998 traces.
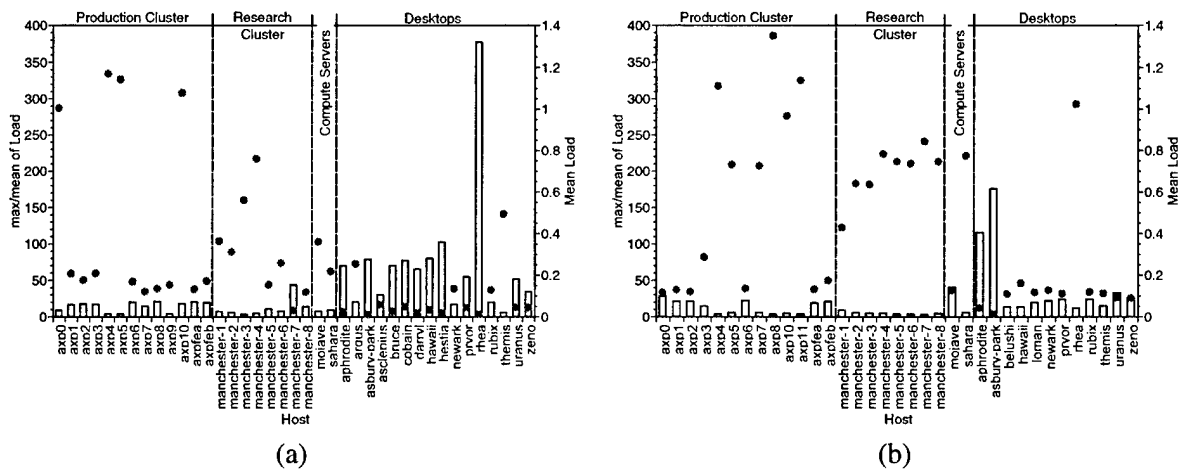
Figure 3.6: Maximum to mean load ratios and mean load: (a) August, 1997 traces, (b) March, 1998 traces.

sures of variability is that lightly loaded (low mean load) hosts are not always preferable over heavily loaded hosts. For example, if the performance metric is itself a relative one (that the running time not vary much relative to the mean running time, say), then a more heavily loaded host may be preferable.

3.4.2   Distributions

We next treated each trace as a realization of an independent, identically distributed (IID) stochastic process. Such a process is completely described by its probability distribution function (pdf), which does not change over time. Since we have a vast number of data points for each trace, histograms closely approximate this underlying pdf. We examined the histograms of each of our load traces and fitted normal and exponential distributions to them. To illustrate the following discussion, Figure 3.7 shows the histograms of load measurements on (a) axp0 and (b) axp7 on August 19, 1997 (86400 samples each). Axp0 has a high mean load, while axp7 is much more lightly loaded.

Some of the traces, especially those with high mean loads, have multi-modal histograms. Figure 3.7(a) is an example of such a multi-modal distribution while Figure 3.7(b) shows a unimodal distribution. Typically,

(a) axp0   (b) axp7
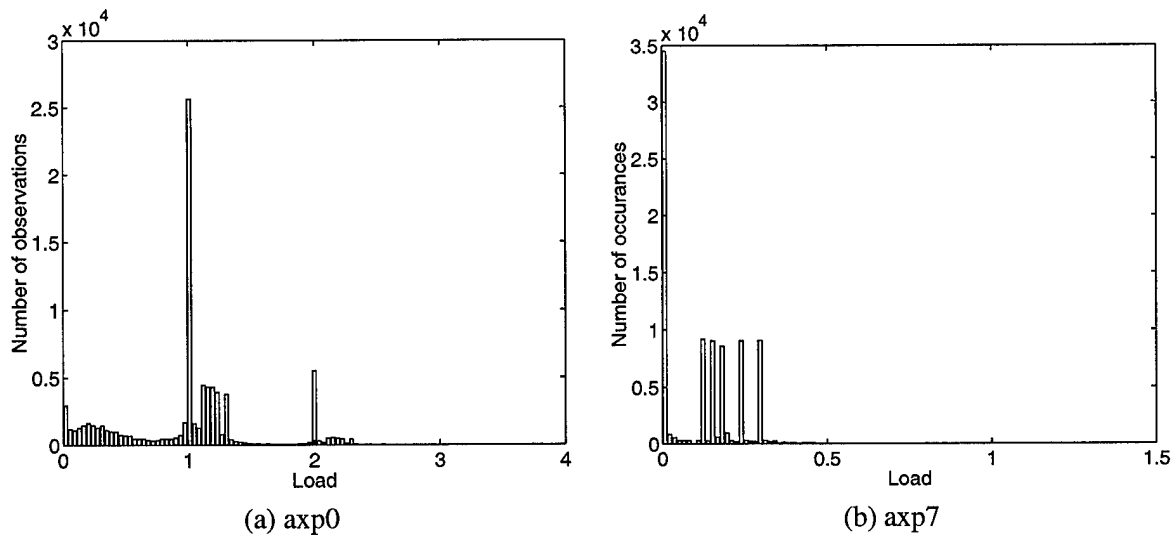
Figure 3.7: Histograms for load on axp0 and axp7 on August 19, 1997.

(a) axp7 versus normal   (b) axp7 versus exponential
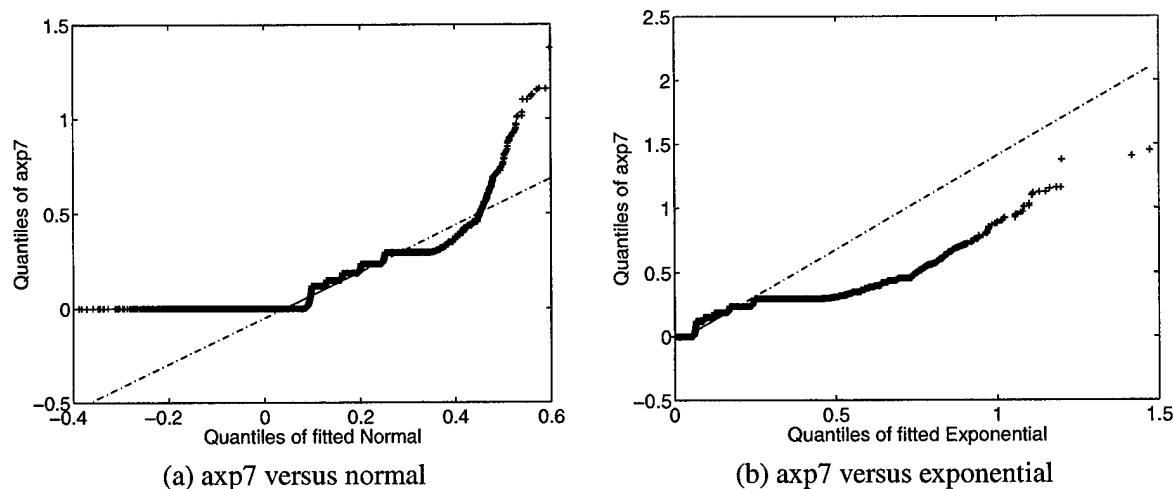
Figure 3.8: Quantile-quantile plots for axp7 trace of August 19, 1997.

the modes are integer multiples of 1.0 (and occasionally 0.5). One explanation for this behavior is that jobs on these machines are for the most part compute bound and thus the ready queue length corresponds to the number of jobs. This seems plausible for the cluster machines, which run scientific workloads for the most part. However, such multi-modal distributions were also noticed on the some of the other machines.

The rough appearance of the histograms (consider Figure 3.7(b)) is due to the fact that the underlying quantity being measured (ready queue length) is discrete. Load typically takes on 600-3000 unique values in these traces. Shannon's entropy measure [121] indicates that the load traces can be encoded in 1.4 to 8.5 bits per value, depending on the trace. These observations and the histograms suggest that load spends most of its time in one of a small number of levels.

The histograms share very few common characteristics and did not conform well to the analytic distributions we fit to them. Quantile-quantile plots are a powerful way to assess how a distribution fits data (cf. [66, pp. 196–200]) The quantiles (the α quantile of a pdf (or histogram) is the value x at which 100α % of the probability (or data) falls to the left of x) of the data set are plotted against the quantiles of the

hypothetical analytic distribution. Regardless of the choice of parameters, the plot will be linear if the data fits the distribution.

We fitted normal and exponential distributions to each of the load traces. The fits are atrocious for the multimodal traces, and we do not discuss them here. For the unimodal traces, the fits are slightly better. Figure 3.8 shows quantile-quantile plots for (a) normal and (b) exponential distributions fitted to the unimodal axp7 load trace of Figure 3.7(b). Neither the normal or exponential distribution correctly captures the tails of the load traces. This can be seen in the figure. The quantiles of the data grow faster than those of the normal distribution toward the right sides of Figure 3.8(a). This indicates that the data has a longer or heavier tail than the normal distribution. Conversely, the quantiles of the data grow more slowly than those of the exponential distribution, as can be seen in Figures 3.8(b). This indicates that the data has a shorter tail than the exponential distribution. Notice that the exponential distribution goes as e^{-x} while the normal distribution goes as e^{-x^2}.

There are two implications of these complex distributions. First, simulation studies and analytic results predicated on simple, analytic distributions may produce erroneous results. Clearly, trace-driven simulation studies are to be preferred. The second implication is that prediction algorithms should not only reduce the overall variance of the load signal, but also produce errors that are better fit an analytic distribution. One reason for this is to make confidence intervals easier to compute.

3.4.3 Time series analysis

We examined the autocorrelation function, partial autocorrelation function, and periodogram of each of the load traces. These time series analysis tools show that past load values have a strong influence on future load values. For illustration, Figure 3.9 shows (a) the axp7 load trace collected on August 19, 1997, (b) its autocorrelation function to a lag of 600, (c) its periodogram, and (d) its partial autocorrelation function to a lag of 600. The analysis of this trace is representative of our results.

The autocorrelation function, which ranges from -1 to 1, shows how well a load value at time t is linearly correlated with its corresponding load value at time $t + \Delta$—in effect, how well the value at time t linearly predicts the value at time $t + \Delta$. Autocorrelation is a function of Δ, and in Figure 3.9(b) we show the results for $0 \leq \Delta \leq 600$. Notice that even at $\Delta = 600$ seconds, values are still strongly correlated. This very strong, long range correlation is common to each of the load traces.

The partial autocorrelation function shows how well purely autoregressive linear models capture the correlation structure of a sequence [23, pp. 64–69]. The square of the value of the function at a lag k indicates the benefit of advancing from a $k - 1$-th order model to a k-th order model. Intuitively, if a k-th order model were sufficient, then the partial autocorrelation function would be zero beyond a lag of k and the autocorrelation function would be infinite. In a dual manner, if a k-th order purely moving average model were sufficient, then the autocorrelation function would be zero beyond a lag of k and the partial autocorrelation function would be infinite. As we can see from Figure 3.9(b) and (d), both functions have extremely large extents. This suggests that mixed models, which combine autoregressive and moving average components are appropriate for modelling host load.

The periodogram of a load trace is the magnitude of the Fourier transform of the load data, which we plot on a log scale (Figure 3.9(c)). The periodogram shows the contribution of different frequencies (horizontal axis) to the signal. What is clear in the figure, and is true of all of the load traces, is that there are significant contributions from all frequencies—the signal looks much like noise. We believe the two noticeable peaks to be artifacts of the kernel sample rate—the kernel is not sampling the length of the ready queue frequently enough to avoid aliasing. Only a few of the other traces exhibit the smaller peaks, but they all share the broad noise-like appearance of this trace.

There are several implications of this time series analysis. First, the existence of such strong autocorre-
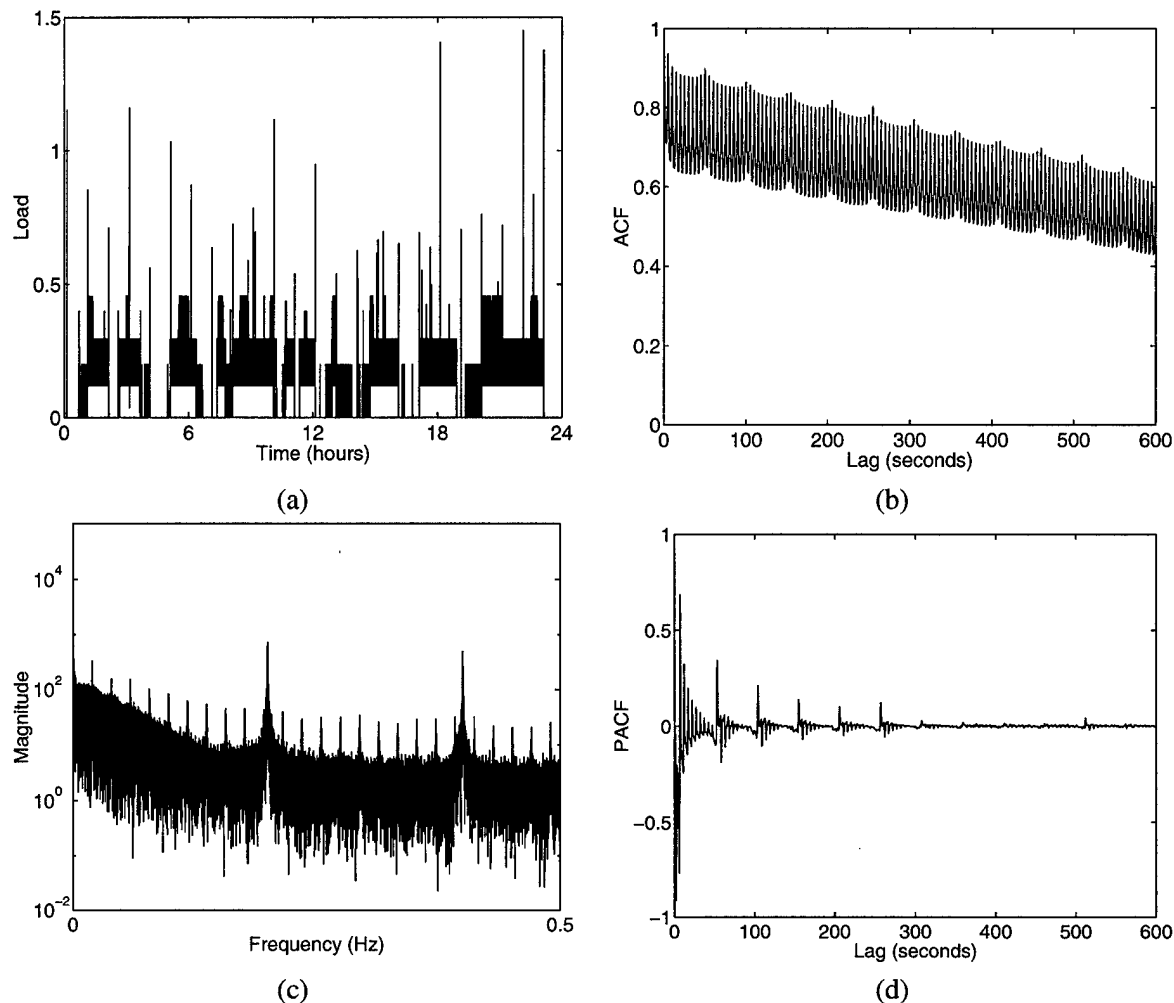
Figure 3.9: Time series analysis of axp7 load trace collected on August 19, 1997: (a) Load trace, (b) Autocorrelation function to lag 600 (10 minutes), (c) Partial autocorrelation function to lag 600 (10 minutes), (d) Periodogram.

lation implies that load prediction based on past load values is feasible. Furthermore, it suggests that linear time series models may be appropriate for this prediction. Such models attempt to parsimoniously capture non-zero autocorrelation functions. The existence of the strong autocorrelation also suggests that simulation models and analytical work that eschews this very clear dependence may be in error. Finally, the almost noise-like periodograms suggest that quite complex, possibly nonlinear models will be necessary to produce or predict load.

3.5 Self-similarity

The key observation discussed in this section is that each of the load traces exhibits a high degree of self-similarity. This is significant for two reasons. First, it means that load varies significantly across all time-scales—it is not the case that increasing smoothing of the load quickly tames its variance. A job will have a great deal of variance in its running time regardless of how long it is. Second, it suggests that load is difficult to model and predict well. In particular, self-similarity is indicative of long memory, possibly non-
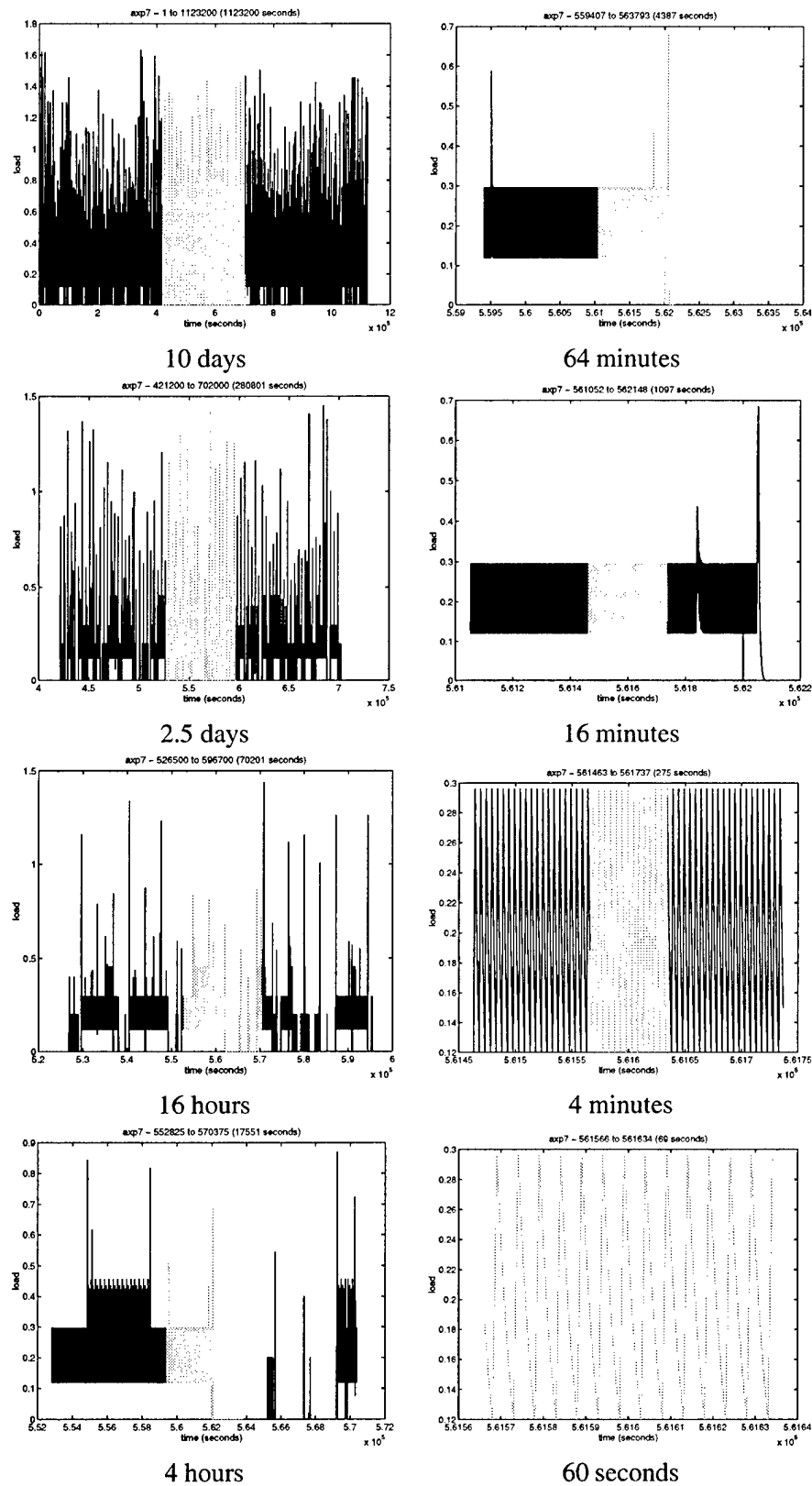
Figure 3.10: Visual representation of self-similarity. Each graph plots load versus time and "zooms in" on the middle quarter
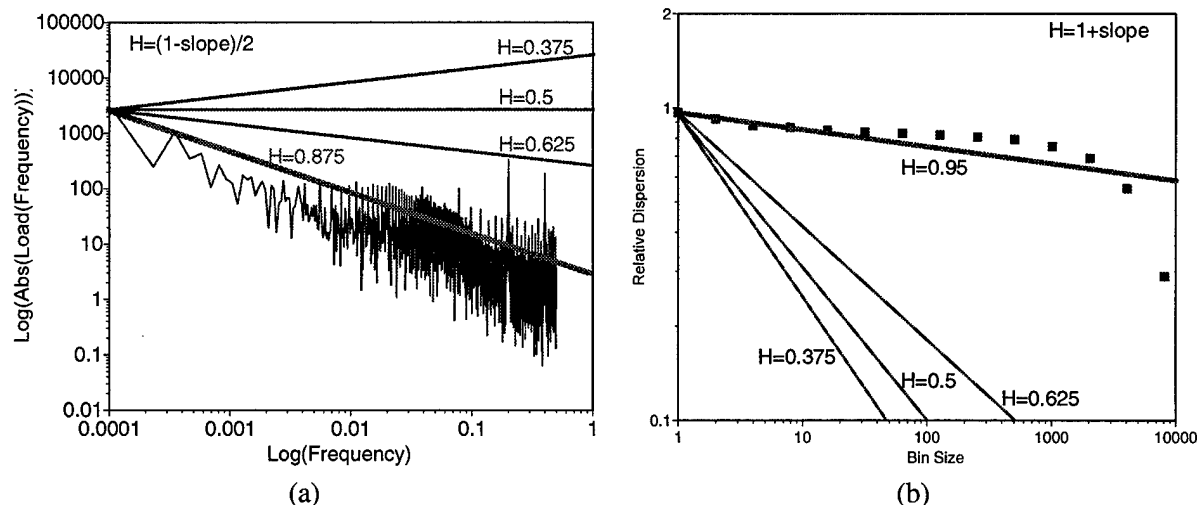
Figure 3.11: Meaning of the Hurst parameter: (a) Frequency domain interpretation using power spectral analysis, (b) Effect of increased smoothing using dispersional analysis.

stationary stochastic processes such as fractional ARIMA models [63, 55, 15], and fitting such models to data and evaluating them can be quite expensive.

Figure 3.10 visually demonstrates the self similarity of the axp7 load trace. The top left graph in the figure plots the load on this machine versus time for 10 days. Each subsequent graph "zooms in" on the highlighted central 25% of the previous graph, until we reach the bottom right graph, which shows the central 60 seconds of the trace. The plots are scaled to make the behavior on each time scale obvious. In particular, over longer time scales, wider scales are necessary. Intuitively, a self-similar signal is one that looks similar on different time scales given this rescaling. Although the behavior on the different graphs is not identical, we can clearly see that there is significant variation on all time scales.

An important point is that as we smooth the signal (as we do visually as we "zoom out" toward the top of the page in Figure 3.10), the load signal strongly resists becoming uniform. This suggests that low frequency components are significant in the overall mix of the signal, or, equivalently, that there is significant long range dependence. It is this property of self-similar signals that most strongly differentiates them and causes significant modeling difficulty.

Self-similarity is more than intuition—it is a well defined mathematical statement about the relationship of the autocorrelation functions of increasingly smoothed versions of certain kinds of long-memory stochastic processes. These stochastic processes model the sort of the mechanisms that give rise to self-similar signals. We shall avoid a mathematical treatment here, but interested readers may want to consult [78] or [91] for a treatment in the context of networking or [11] for its connection to fractal geometry, or [15] for a treatment from a linear time series point of view. Interestingly, self-similarity has revolutionized network traffic modelling in the 1990s [50, 78, 91, 138].

The degree and nature of the self-similarity of a sequence is summarized by the Hurst parameter, H [65]. Intuitively, H describes the relative contribution of low and high frequency components to the signal. Consider Figure 3.11(a), which plots the periodogram (the magnitude of the Fourier transform) of the axp7 load trace of August 19, 1997 on a log-log scale. In this transformed form, we can describe the trend with a line of slope $-\beta$ (meaning that the periodogram decays hyperbolically with frequency ω as $\omega^{-\beta}$. The Hurst parameter H is then defined as $H = (1 + \beta)/2$. As we can see in Figure 3.11(a), $H = 0.5$ corresponds to a line of zero slope. This is the uncorrelated noise case, where all frequencies make a roughly equal contribution. As H increases beyond 0.5, we see that low frequencies make more of a contribution. Similarly,
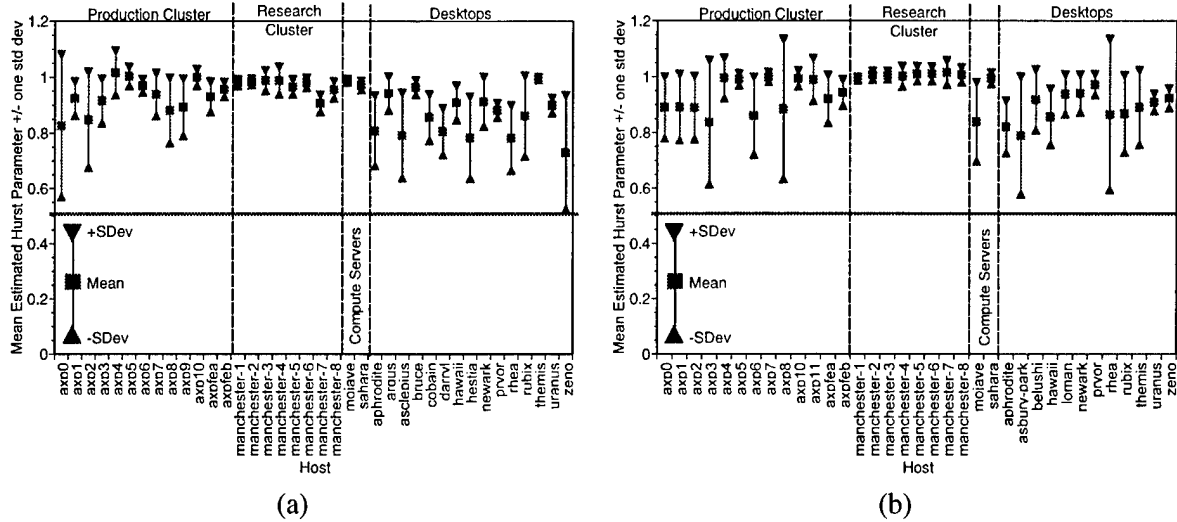
Figure 3.12: Hurst parameter estimates (mean of four point estimates and standard deviation) : (a) August, 1997 traces, (b) March, 1998 traces.

as H decreases below 0.5, low frequencies make less of a contribution. $H > 0.5$ indicates self-similarity with positive near neighbor correlation, while $H < 0.5$ indicates self-similarity with negative near neighbor correlation. Figure 3.11(a) shows that the axp7 trace is indeed self-similar with $H = 0.875$. This method of determining the Hurst parameter is known as power spectral analysis.

Figure 3.11(b) illustrates another way to think about the Hurst parameter H. To create the figure, we binned the load trace with increasingly larger, non-overlapping bins and then plotted the relative dispersion (the standard deviation normalized by the mean) of the binned series versus the size of the bins. For example, at a bin size of 8 we averaged the first 8 samples of the original series to form the first sample of the binned series, the next 8 to form the second sample, and so on. The figure shows that the relative dispersion of this new 8-binned series is slightly less than one. If the original load trace is self-similar, the relative dispersion should decline hyperbolically with increasing bin size. On a log-log scale such we use in the figure this relationship would appear to be linear with a slope of $H - 1$. We see that this is indeed the case for the axp7 trace. Notice that this method for estimating H, which is called dispersional analysis, gives a slightly different H (0.95) than power spectral analysis. What is important is that in both figures we see a hyperbolic relationship, and that both estimates for H are much larger than 0.5.

We examined each of the load traces for self-similarity and estimated each one's Hurst parameter. There are many different estimators for the Hurst parameter [130], but there is no consensus on how to best estimate the Hurst parameter of a measured series. The most common technique is to use several Hurst parameter estimators and try to find agreement among them. The four Hurst parameter estimators we used were R/S analysis, the variance-time method, dispersional analysis, and power spectral analysis. A description of these estimators as well as several others may be found in [11]. The advantage of these estimators is that they make no assumptions about the stochastic process that generated the sequence. However, they also cannot provide confidence intervals for their estimates. Estimators such as the Whittle estimator [15] can provide confidence intervals, but a stochastic process model must be assumed over which an H can be found that maximizes its likelihood.

We implemented the estimators using Matlab and validated each one by examining degenerate series with known H and series with specific H generated using the random midpoint displacement method. The dispersional analysis method was found to be rather weak for H values less than about 0.8 and the power spectral analysis method gave the most consistent results.
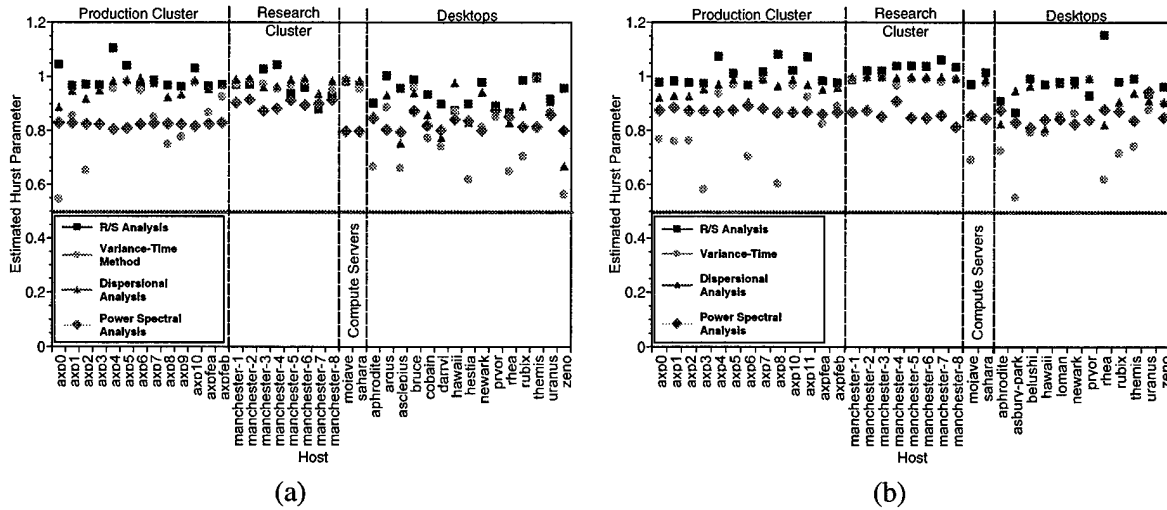
Figure 3.13: Hurst parameter point estimates: (a) August, 1997 traces, (b) March, 1998 traces.

Figure 3.12 presents our estimates of the Hurst parameters of each of load traces. In the graph, each central point is the mean of the four estimates, while the outlying points are at +/- one standard deviation. Figure 3.13 shows the four actual point estimates for each trace. Notice that for small H values, the standard deviation is high due to the inaccuracy of dispersional analysis. The important point is that the mean Hurst estimates are all significantly above the $H = 0.5$ line and their +/- one standard deviation points are also above the line.

The traces exhibit self-similarity Hurst parameters ranging from 0.73 to 0.99, with a strong bias toward the top of that range. An examination of the correlation coefficients (CCs) in Figure 3.2 shows some surprising results. For the August, 1997 traces, the Hurst parameter is positively correlated with mean load (CC=0.45) and the standard deviation of load (CC=0.58), but is negatively correlated with the max/mean load ratio (CC=-0.49). On the other hand, for the March, 1998 traces, the Hurst parameter is negatively correlated with mean load (CC=-0.30) and the standard deviation of load (CC=-0.41), but is positively correlated with the max/mean ratio (CC=0.36). Furthermore, in the March, 1998 traces, we find the Hurst parameter is strongly positively correlated with the epoch statistics, which is not the case at all for the August, 1997 traces. It is not clear what the cause for this difference between the traces is.

As we discussed above, self-similarity has implications for load modeling and for load smoothing. The long memory stochastic process models that can capture self-similarity tend to be expensive to fit to data and evaluate. Smoothing the load (by mapping large units of computations instead of small units, for example) may be misguided since variance may not decline with increasing smoothing intervals as quickly as quickly as expected. Consider smoothing load by averaging over an interval of length m. Without long range dependence ($H = 0.5$), variance would decay with m as $m^{-1.0}$, while with long range dependence, as m^{2H-2} or $m^{-0.54}$ and $m^{-0.02}$ for the range of Hurst parameters we measured.

3.6 Epochal behavior

The key observation discussed in this section is that while load varies in complex ways, the manner in which it changes remains relatively constant for relatively long periods of time. We refer to a period of time in which this stability holds true as an epoch. For example, the load signal could be a 0.25 Hz Sin wave for a minute and a 0.125 Hz sawtooth wave the next minute—each minute is an epoch. That these epochs exist and are long is significant because it suggests that modeling load can be simplified by modeling epochs

separately from modeling the behavior within an epoch. Similarly, it suggests a two stage prediction process.

The spectrogram representation of a load trace immediately highlights the epochal behavior we discuss in this section. A spectrogram combines the frequency domain and time domain representations of a time series. It shows how the frequency domain changes locally (for a small segment of the signal) over time. For our purposes, this local frequency domain information is the "manner in which [the load] changes" to which we referred earlier. To form a spectrogram, we slide a window of length w over the series, and at each offset k, we Fourier-transform the w elements in the window to give us w complex Fourier coefficients. Since our load series is real-valued, only the first $w/2$ of these coefficients are needed. We form a plot where the x coordinate is the offset k, the y coordinate is the coefficient number, $1, 2, \ldots, w/2$ and the z coordinate is the magnitude of the coefficient. To simplify presentation, we collapse to two dimensions by mapping the logarithm of the z coordinate (the magnitude of the coefficient) to color. The spectrogram is basically a midpoint in the tradeoff between purely time-domain or frequency-domain representations. Along the x axis we see the effects of time and along the y axis we see the effects of frequency.

Figure 3.14 shows a representative case, a 24 hour trace from the PSC host axp7, taken on August 19, 1997. The top graph shows the time domain representation, while the bottom graph shows the corresponding spectrogram representation. What is important to note (and which occurs in all the spectrograms of all the traces) are the relatively wide vertical bands. These indicate that the frequency domain of the underlying signal stays relatively stable for long periods of time. We refer to the width of a band as the duration of that frequency epoch.

That these epochs exist can be explained by programs executing different phases, programs being started and shut down, and the like. The frequency content within an epoch contains energy at higher frequencies because of events that happen on smaller time-frames, such as user input, device driver execution, and daemon execution.

We can algorithmically find the edges of these epochs by computing the difference in adjacent spectra in the spectrogram and then looking for those offsets where the differences exceed a threshold. Specifically, we compute the sum of mean squared errors for each pair of adjacent spectra. The elements of this error vector are compared to an epsilon (5% here) times the mean of the vector. Where this threshold is exceeded, a new epoch is considered to begin. Having found the epochs, we can examine their statistics. Figure 3.15 shows the mean epoch length and the +/- one standard deviation levels for each of the load traces. The mean epoch length ranges from about 150 seconds to over 450 seconds, depending on which trace. The standard deviations are also relatively high (80 seconds to over 600 seconds). It is the Production Cluster class which is clearly different when it comes to epoch length. The machines in this class tend to have much higher means and standard deviations than the other machines. One explanation might be that most of the machines run batch-scheduled scientific jobs which may well have longer computation phases and running times. However, two of the interactive machines also exhibit high means and standard deviations. Interestingly, there is no correlation of the mean epoch length and standard deviation to the mean load for either set of traces (Figure 3.2). However, for the March, 1998 traces, we find correlations between the epoch length statistics and the Hurst parameter that are particularly strong. It is not clear why this difference exists between the traces.

The standard deviations of epoch length in Figure 3.15 give us an absolute measure of the variance of epoch length. Figure 3.16 shows the coefficient of variance (COV) of epoch length and mean epoch length for each trace. The COV is our relative measure of epoch length variance. Unlike with load (Section 3.4), these absolute and relative measures of epoch length variance are *both* positively correlated with the mean epoch length. In addition, the correlation is especially strong (the CCs are at least 0.88). As epoch length increases, it varies more in both absolute and relative terms. The statistical properties of epoch lengths are independent of the statistical properties of load.

The implication of long epoch lengths is that the problem of predicting load may be able to be decom-
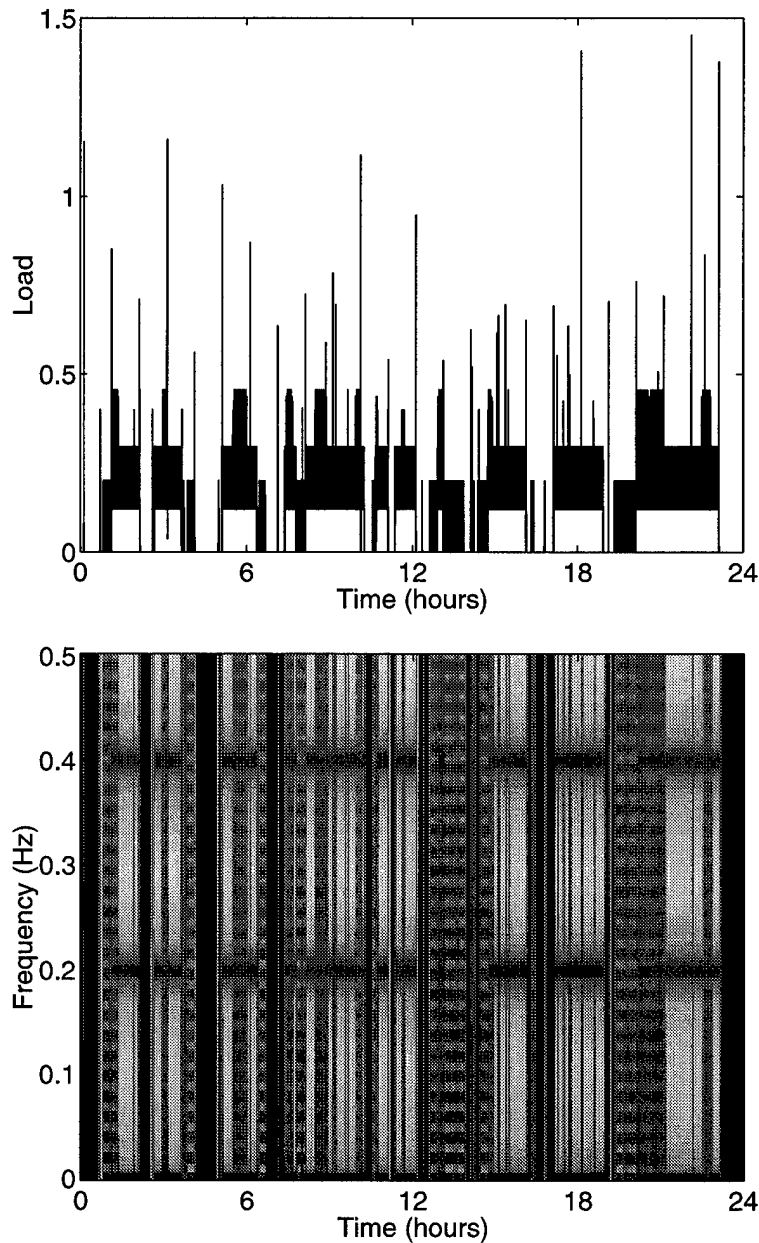
Figure 3.14: Time domain and spectrogram representations of load for host axp7 on August 19, 1997.

posed into a segmentation problem (finding the epochs) and a sequence of smaller prediction subproblems (predicting load within each epoch).

Strictly speaking, epochal behavior means that load is not stationary. However, it is also not free to wander at will—clearly load cannot rise to infinite levels or fall below zero. This is compatible with the "borderline stationarity" implied by self-similarity. It is also important to note that the nonstationarity is not "smooth" and therefore is difficult to model with an integration process, such as in an ARIMA or ARFIMA model.
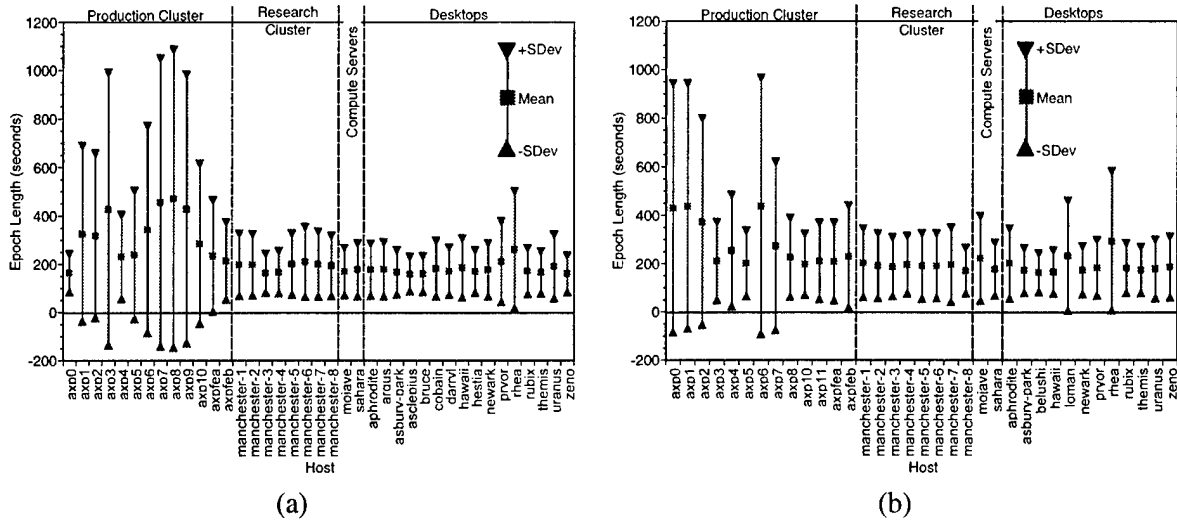
Figure 3.15: Mean epoch length +/- one standard deviation: (a) August, 1997 traces, (b) March, 1998 traces.
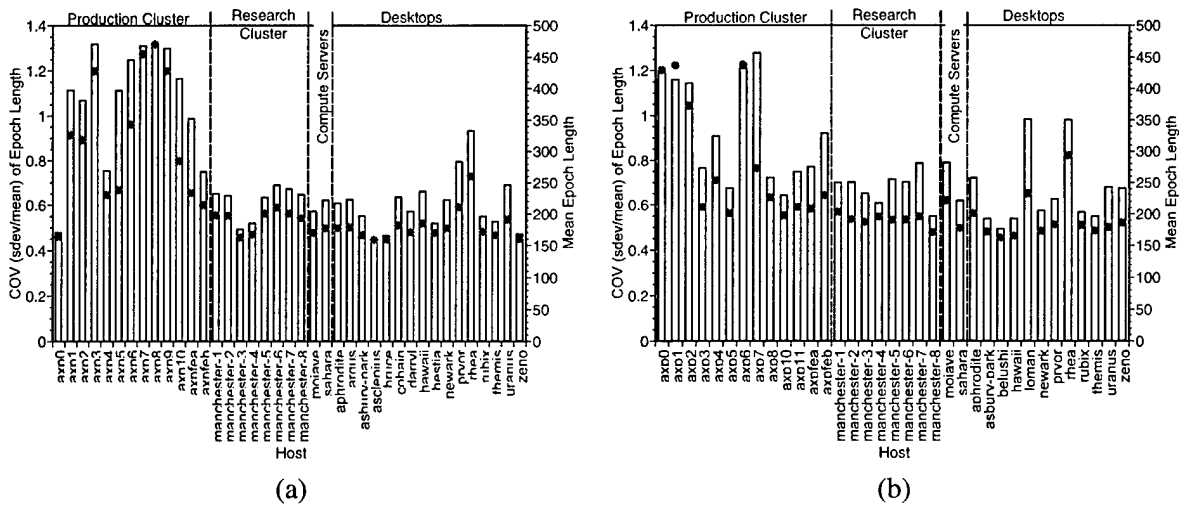
Figure 3.16: COV of epoch length and mean epoch length: (a) August, 1997 traces, (b) March, 1998 traces.

3.6.1 Lack of seasonality

The epochal behavior that the load traces exhibit is not the same thing as seasonality in the time series analysis sense [25, 23]. Seasonality means that there are dominant (or at least visible) underlying periodic signals on top of which are layered other signals. It is not unreasonable to expect seasonality given that other studies [92] have found that availability of compute resources to change regularly over the hours of the working day and the days of the working week. However, examination of the power spectrums and autocorrelations of the load traces suggests that load does *not* exhibit seasonality. We feel this does not contradict the earlier results—the fluctuation of resources simply is not sufficiently periodic to qualify as seasonality in the strict time series sense.

3.7 Conclusions

This chapter has applied the first four steps of the resource signal methodology to host load signals. We identified host load as a powerful way to measure CPU availability, developed a sensor for this signal, and determined an appropriate sample rate. We collected long, fine grain load measurements on a wide variety of machines at two different times of the year. The results of an extensive statistical analysis of these traces and their implications are the following:

(1) The traces exhibit low means but very high standard deviations and maximums. Relatively few of the traces had mean loads of 1.0 or more. The standard deviation is typically at least as large as the mean, while the maximums can be as much as two orders of magnitude larger. The implication is that these machines have plenty of cycles to spare to execute jobs, but the running time of these jobs will vary drastically.

(2) Standard deviation and maximum, which are absolute measures of variation, are positively correlated with the mean, so a machine with a high mean load will also tend to have a large standard deviation and maximum. However, these measures do not grow as quickly as the mean, so their corresponding relative measures actually *shrink* as the mean increases. The implication is that if the goal in scheduling a task is to optimize a relative performance metric, it may not be unreasonable to use the host with higher mean load.

(3) The traces have complex, rough, and often multi-modal distributions that are not well fitted by analytic distributions such as the normal or exponential distributions. Even for the traces which exhibit unimodally distributed load, the normal distribution's tail is too short while the exponential distribution's tail is too long. The implication is that modeling and simulation that assumes convenient analytical load distributions may be flawed.

(4) Time series analysis of the traces shows that load is strongly correlated over time. The autocorrelation function typically decays very slowly while the periodogram shows a broad, almost noise-like combination of all frequency components. An important implication is that history-based load prediction schemes, such as linear time series models, seem very feasible. However, the complex frequency domain behavior suggests that linear modeling schemes may have difficulty. From a modeling point of view, it is clearly important that these dependencies between successive load measurements are captured.

(5) The traces are self-similar. Their Hurst parameters range from 0.73 to 0.99, with a strong bias toward the top of that range. This tells us that load varies in complex ways on all time scales and is long term dependent. This has several important implications. First, smoothing load by averaging over an interval results in much smaller decreases in variance than if load were not long range dependent. This suggests that task migration in the face of adverse load conditions may be preferable to waiting for the adversity to be ameliorated over the long term. The self-similarity result also suggests certain modeling approaches, such as fractional ARIMA models [63, 55, 15] which can capture this property.

(6) The traces display epochal behavior. The local frequency content of the load signal remains quite stable for long periods of time (150-450 seconds mean) and changes abruptly at the boundaries of such epochs. This suggests that the problem of predicting load may be able to be decomposed into a sequence of smaller subproblems. In particular, when using linear models, none of which can capture this sort of "rough" nonstationarity, it may be necessary to refit the models at epoch boundaries.

Given these properties, we decided to study the performance of Box-Jenkins linear time series models [23] and fractional ARFIMA models [55, 63, 15] for short range prediction of host load. The next chapter describes how we applied the fifth step of the resource signal methodology to find which of these models was most appropriate. The final step, implementing an on-line host load prediction system based on the appropriate model was easy using the RPS Toolkit.

Chapter 4

Host Load Prediction

This dissertation argues for basing real-time scheduling advisors on explicit resource-oriented prediction, specifically on the prediction of resource signals. The signal we focus on is host load. In the last chapter, we studied the statistical properties of a large collection of host load traces and came to the conclusion that linear time series models might be appropriate for predicting host load. This carried out the first four steps of our resource signal methodology, as presented in Section 2.1. This chapter carries out the final two steps of the methodology. We use the RPS Toolkit to evaluate many different linear models on the load traces we described in the previous chapter and come to the conclusion that autoregressive models of order 16 or better are appropriate for predicting the host load signal. Having decided on a model, implementing an RPS-based on-line host load prediction system that uses it is trivial, but the evaluation of that system is not, as we shall see in the subsequent chapters.

It is important to note that a prediction consists not only of the expected future values of the discrete-time host load signal, but also the expected variance (or mean squared error) of their prediction errors and the covariance of the errors with each other. In other words, each prediction is qualified with an estimate of how erroneous it can be. These measures of prediction quality provide the basis of statistical reasoning using the predictions. In the next chapter, we use the predictions and the estimates of their quality to compute confidence intervals for running times of tasks.

Intuitively, the expected mean squared error and the covariances are measures of the "surprise" the user of the predictions is likely to encounter. The mean squared error is directly comparable to the variance of the signal itself, while the covariances can be compared to the autocorrelation of the load signal. If we want to predict a future value of the signal, then the mean squared error is sufficient to qualify the prediction. If the goal is to predict some function of a number of future values (the average over an interval of time, for example), then the covariances are needed because the prediction errors of the individual values are not necessarily independent. This chapter focuses on the measured mean squared error of the predictors. Ideally, the prediction errors will be less surprising (have lower mean square error) than the signal itself.

The estimates of the mean squared error of the predictions are derived from the fit of the predictive model to some region of the signal. However, in prediction, we are concerned with the measured mean square error of the model on the signal subsequent to where it was fit. In other words, we don't care how well the model fits the signal, but in how well the fitted model, when used in practice, predicts the signal. For this reason, this chapter concentrates on the measured mean square error of the different predictors.

In a system such as RPS, predictive models are likely to be refit at arbitrary times whenever monitoring software decides that their performance has degraded. Because of this, it is important to consider, in as unbiased a fashion as possible, the performance of the models in different situations. Our evaluation of the predictive models is randomized in order to avoid bias. Another important point is that we are interested in models that not only have a low measured mean squared error on average, but whose measured mean

squared error is consistently low. What this requirement for consistent predictability means is that we desire that, regardless of when we fit the model, we can expect it to provide good results.

We find that load is, in fact, consistently predictable to a very useful degree from past behavior, and that simple linear time series models, especially AR(16)s or better, are sufficiently powerful load predictors. These results are surprising given the complex behavior of host load that we identified in the previous chapter. As far as we are aware, this is the first study to identify these properties of host load and then to evaluate the practical predictive power of linear time series models that both can and cannot capture them. It is also unique in focusing on predictability on the scale of seconds as opposed to minutes or longer time scales.

4.1 Prediction

Recall from the previous chapter that we write the host load signal as $\langle z_t \rangle = \ldots, z_{t-1}, z_t, z_{t+1}, \ldots$, where t is the current time and z_{t+j} denotes a sample of the signal j steps into the future. The *prediction* of any particular load sample will be written with a hat. For example, $\hat{z}_{t+1}$ denotes a prediction of the value z_{t+1} (the one-step-ahead prediction) using all previous values of the signal. Such predictions use all available data up to and including time t. We will also write $\hat{z}_{t+i,t+j}$ to denote the prediction of z_{t+j} using all values up to and including z_{t+i}.

An RPS-based resource prediction system provides one-step-ahead to m-step-ahead predictions, where m is user-specified. At time $t + i$, the value z_{t+i} is measured and pushed to RPS, which responds with the vector $[\hat{z}_{t+i,t+i+1}, \hat{z}_{t+i,t+i+2}, \ldots, \hat{z}_{t+i,t+i+m}]$. When z_{t+i+1} becomes available, we measure its one-step-ahead prediction error as $a_{t+i,t+i+1} = z_{t+i+1} - \hat{z}_{t+i,t+i+1}$. Similarly, when z_{t+i+2} becomes available, we can compute the two-step-ahead prediction error as $a_{t+i,t+i+2} = z_{t+i+2} - \hat{z}_{t+i,t+i+2}$. Taken over all i, we summarize the set of one-step-ahead prediction errors by its variance, which is commonly called the one-step-ahead *mean squared error*. The 2-step-ahead mean squared error is computed similarly. There is a mean squared error associated with every *lead time* k. We shall study the one- to 30-step-ahead (or one- to 30-second-ahead, given our sample rate of 1 Hz) mean squared errors.

It is important to note that the mean k-step-ahead prediction error should be zero (which it is in this study). A non-zero mean would indicate that the predictor has an unhealthy systematic bias. The mean squared error is the important quantity to examine, since it measures how far a particular prediction error is likely to vary from zero. With a good predictor, the sequence of k-step-ahead prediction errors should be IID. If the error distribution is normal, then the k-step-ahead mean squared error is sufficient to place confidence intervals on the predictions. We found that the prediction errors are indeed independent, but not normally distributed. In the next chapter we show that making the normality assumption nonetheless leads to good results in predicting the running time of tasks.

While the sequence of k-step-ahead prediction errors are independent, the $k + 1$-step-ahead prediction error is dependent on the k-step-ahead prediction error. As we show in the next chapter, this becomes important when it is necessary to sum over a number of predictions, say over a time horizon from now to 10 seconds from now. We do not evaluate the covariance of the prediction errors in this chapter. It is unclear what the figure of merit should be. The more predictable a signal is, the greater the covariances typically are.

4.2 Predictive models

For the simplest prediction, the long-term mean of the signal, the mean squared error (for any lead time) is simply the variance of the load signal. As we saw in the previous chapter, load has quite high variance and

exhibits other complex properties. The hope is that more sophisticated prediction schemes will have much better mean squared errors.

On the one hand, the analysis of the last chapter suggested that linear time series models such as those in the Box-Jenkins [23] AR, MA, ARMA, and ARIMA classes might be appropriate for predicting load. On the other hand, the existence of self-similarity induced long-range dependence suggested that such models might require an impractical number of parameters or that the much more expensive ARFIMA model class, which explicitly captures long-range dependence, might be more appropriate. Since it is not obvious which model is best, we empirically evaluated the predictive power of the AR, MA, ARMA, ARIMA, and ARFIMA model classes, as well as a simple ad hoc windowed-mean predictor called BM and a long-term mean predictor called MEAN. We also looked at the performance of BM(1) models, in which the last measured value of the signal is used as the prediction of all future values. We call this the LAST model class. These model classes, their implementations, and their overheads are described in Chapter 2.

It is important to note that host load is an exponentially smoothed signal—an AR(1) with $\phi_1 = 0.8187$. The benefits of higher order linear models, as well as the clear benefits of the linear models over LAST and BM show that there is significant predictability in the load signal beyond that introduced by this exponential smoothing. Furthermore, the better models show significantly better performance for long lead times than would be expected from simply the exponential smoothing. Also, if we factor out the exponential smoothing, we still see significant predictability. Finally, as we show in the next chapter, our predictions of the unmolested load signal work well to estimate good confidence intervals for the running time of tasks.

4.3 Evaluation methodology

Our methodology is designed to determine whether there are consistent differences in the *practical predictive power* of the different model classes. Other goals are also possible. For example, one could determine the *explanatory power* of the models by evaluating how well they fit data, or the *generative power* of the model by generating new load traces from fitted models and evaluating how well their statistical properties match the original traces. We have touched on these other goals, but do not discuss them here. To assess the practical predictive power of the different model classes, we designed a randomized, trace-driven simulation methodology that fits randomly selected models to subsequences of a load trace and tests them on immediately following subsequences.

In essence, a host load prediction system has one thread of control of the following form:

```
do forever {
    get new load measurement;
    update history;
    if (some criteria) {
        refit model to history;
        make new predictor;
    }
    step predictor;
    sleep for sample interval;
}
```

where `history` is a window of previous load values. Fitting the model to the history is an expensive operation. Once a model is fitted, a predictor can be produced from it. *Stepping* this predictor means informing it of a new load measurement so that it can modify its internal state. This is an inexpensive operation. Prediction requests arrive asynchronously and are serviced using the current state of the predictor. A prediction request that arrives at time $t + i$ includes a lead time k. The predicted load values

| | Fit Interval
5 min...3 hours
(m=600 to 10800 samples) | Test Interval
5 min...3 hours
(n=600 to 10800 samples) |

Load Trace
~one week
345,000 to >1M
samples at 1Hz

$$Z_{t_{cross}-m} \qquad Z_{t_{cross}} \qquad Z_{t_{cross}+n-1}$$

Crossover Point
3 hours, trace length - 3 hours
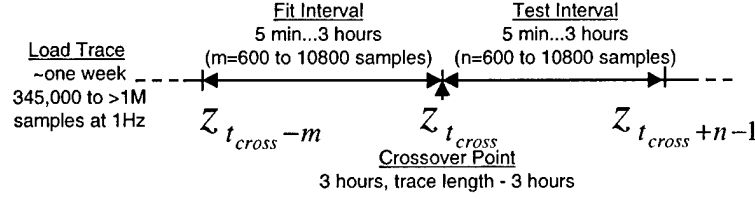
Model class	Number of parameters
MEAN	none
BM(p) (inc. LAST)	p=1..32, chosen by fit
AR(p)	p=1..32
MA(q)	q=1..8
ARMA(p,q)	p=1..4, q=1..4
ARIMA(p,d,q)	p=1..4, d=1..2, q=1..4
ARFIMA(p,d,q)	p=1..4, d by fit, q=1..4

Table 4.1: Testcase generation.

$\hat{z}_{t+i,t+i+1}, \hat{z}_{t+i,t+i+2}, \ldots, \hat{z}_{t+i,t+i+k}$ are returned along with model-based estimates of the mean squared prediction error for each prediction.

Evaluating the predictive power of the different model classes in such a context is complicated because there is such a vast space of configuration parameters to explore. These parameters include: the trace, the model class, the number of parameters we allow the model and how they are distributed, the lead time, the length of the history to which the model is fit, the length of the interval during which the model is used to predict, and at what points the model is refit. We also want to avoid biases due to favoring particular regions of traces.

To explore this space in a reasonably unbiased way, we ran a large number of randomized testcases on each of the traces. Table 4.1 illustrates the parameter space from which testcase parameters are chosen. A testcase is generated and evaluated using the following steps.

1. Choose a random *crossover point*, t_{cross}, from within the trace.

2. Choose a random number of samples, m, from $600, 601, \ldots, 10800$ (5 minutes to three hours). The m samples preceding the crossover point, $z_{t_{cross}-m}, z_{t_{cross}-m+1}, \ldots, z_{t_{cross}-1}$ are in the *fit interval*.

3. Choose a random number of samples, n from $600, 601, \ldots, 10800$ (5 minutes to three hours). The n samples including and following the crossover point, $z_{t_{cross}}, z_{t_{cross}+1}, \ldots, z_{t_{cross}+n-1}$, are in the *test interval*.

4. Choose a random AR, MA, ARMA, ARIMA, or ARFIMA *test model* from the table in Table 4.1, fit it to the samples in the fit interval, and generate a predictor from the fitted test model.

5. For $i = m$ to 1, step the predictor with $z_{t_{cross}-i}$ (the values in the fit interval) to initialize its internal state. After this step, the predictor is ready to be tested.

6. For $i = 0$ to $n - 1$ do the following:

   (a) Step the predictor with $z_{t_{cross}+i}$ (the next value in the test interval).

   (b) For each lead time $k = 1, 2, \ldots, 30$ seconds, produce the predictions
   $\hat{z}_{t_{cross}+i,t_{cross}+i+k}$. $\hat{z}_{t_{cross}+i,t_{cross}+i+k}$ is the prediction of $z_{t_{cross}+i+k}$ given the samples
   $z_{t_{cross}-m}, z_{t_{cross}-m+1}, \ldots, z_{t_{cross}+i}$. Compute the prediction errors
   $a_{t_{cross}+i,t_{cross}+i+k} = \hat{z}_{t_{cross}+i,t_{cross}+i+k} - z_{t_{cross}+i+k}$.

7. For each lead time $k = 1, 2, \ldots, 30$ seconds, analyze the k-step-ahead prediction errors
$a_{t_{cross}+i, t_{cross}+i+k}$, $i = 0, 1, \ldots, n - 1$.

8. Output the testcase parameters and the analysis of the prediction errors.

For clarity in the above, we focused on the linear time series model under test. Each testcase also includes a parallel evaluation of the BM and MEAN models to facilitate direct comparison with the simple BM model and the raw signal variance.

The lower limit we place on the length of the fit and test intervals is purely prosaic—the ARFIMA model needs about this much data to be successfully fit. The upper limit is chosen to be greater than most epoch lengths so that we can see the effect of crossing epoch boundaries. The models are limited to eight parameters because fitting larger MA, ARMA, ARIMA, or ARFIMA models is prohibitively expensive in a real system. We did also explore larger AR models, up to order 32.

The analysis of the prediction errors includes the following. For each lead time, the minimum, median, maximum, mean, mean absolute, and mean squared prediction errors are computed. The one-step-ahead prediction errors (ie, $a_{t_{cross}+i, t_{cross}+i+1}$, $i = 1, 2, \ldots, n$) are also subject to IID and normality tests as described by Brockwell and Davis [25, pp. 34–37]. IID tests included the fraction of the autocorrelations that are significant, the Portmanteau Q statistic (the power of the autocorrelation function), the turning point test, and the sign test. Normality was tested by computing the R^2 value of a least-squares fit to a quantile-quantile plot of the values or errors versus a sequence of normals of the same mean and variance.

Because the properties of the two sets of traces are so similar, our study focused on the August, 1997 set. We implemented the randomized evaluation using the parallelized RPS-based tool described in Chapter 2. We ran approximately 152,000 testcases, which amounted to about 4000 testcases per trace, or about 1000 per model class and parameter set, or about 30 per trace, model class and parameter set. We ran an additional 38,000 testcases explicitly to compare AR(16) models with the LAST model (1000 testcases per trace). It is these testcases that we discuss in the remainder of this chapter. We also ran an additional 152,000 fully randomized testcases on the traces where we separately removed the exponential averaging. Our parallelized simulation discarded testcases in which an unstable model "blew up," (less than 5%, almost always ARIMA or ARFIMA models) either detectably or due to a floating point exception. The results of the accepted testcases were committed to a SQL database to simplify the analysis discussed in the following section.

4.4 Results

The section addresses the following questions: Is load consistently predictable? If so, what are the consistent differences between the different model classes, and which class is preferable? To answer these questions we analyze the database of randomized testcases from Section 4.3. For the most part we will address only the mean squared error results, although we will touch on the other results as well.

4.4.1 Load is consistently predictable

For a model to provide consistent predictability of load, it must satisfy two requirements. First, for the average testcase, the model must have a considerably lower expected mean squared error than the expected raw variance of the load signal (ie, the expected mean squared error of the MEAN model). The second requirement is that this expectation is also very likely, or that there is little variability from testcase to testcase. Intuitively, the first requirement says that the model provides better predictions on average, while the second says that most predictions are close to that average.

Figure 4.1 suggests that load is indeed consistently predictable in this sense. The figure is a Box plot that shows the distribution of one-step-ahead mean squared error measures for 8 parameter models on all
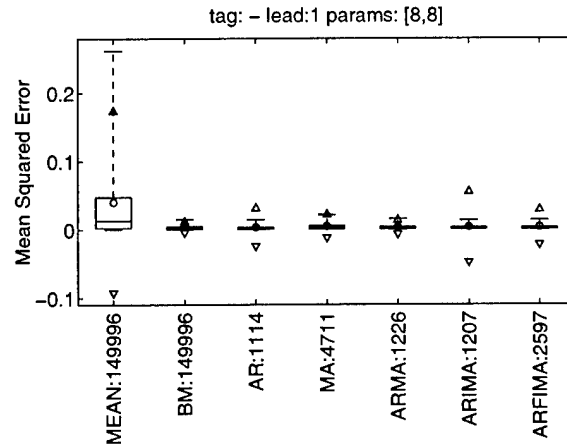
Figure 4.1: All traces, 1 second lead, 8 parameters
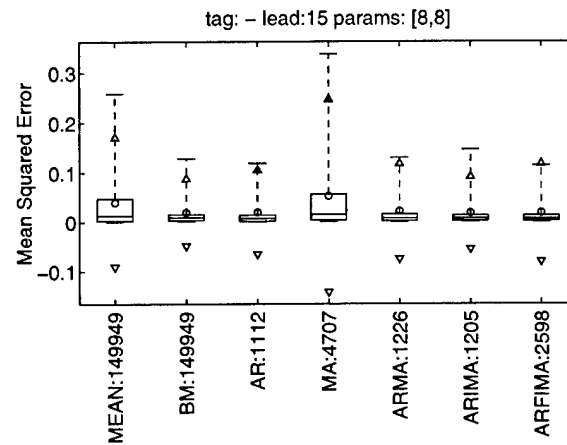
Figure 4.2: All traces, 15 second lead, 8 parameters.

of the traces. In the figure, each category is a specific class of model and is annotated with the number of samples for that class. For each class, the circle indicates the expected mean squared error, while the triangles indicated the 2.5th and 97.5th percentiles assuming a normal distribution. The center line of each box shows the median while the lower and upper limits of the box show the 25th and 75th percentiles and the lower and upper whiskers show the actual 2.5th and 97.5th percentiles.

Notice that the expected raw variance (MEAN) of a testcase is approximately 0.05, while the expected mean squared error for *all* of the model classes is nearly zero. The figure also shows that our second requirement for consistent predictability is met. We see that the variability around the expected mean squared error is much lower for the predictive models than for MEAN. For example, the 97.5th percentile of the raw variance is almost 0.3, while it is about 0.02 for the predictive models.

Figures 4.2 and 4.3 show the results for 15 second predictions and 30 second predictions. Notice that, except for the MA models, the predictive models are consistently better than the raw load variance, even with 30 second ahead predictions. We also see that MA models perform quite badly, especially at higher lead times. This was also the case when we considered the traces individually and broadened the number of

tag: – lead:30 params: [8,8]
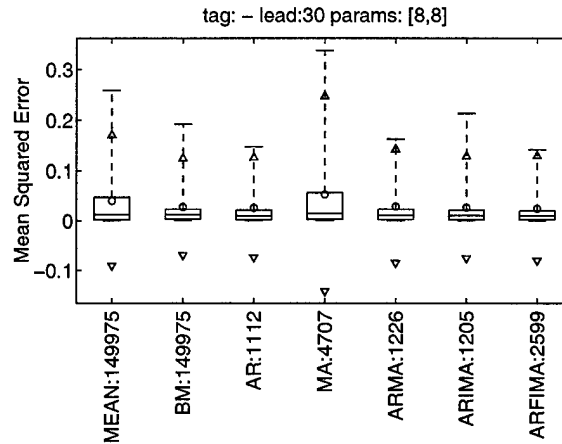
Figure 4.3: All traces, 30 second lead, 8 parameters.

tag: axp0.psc lead:1 params: [8,8]
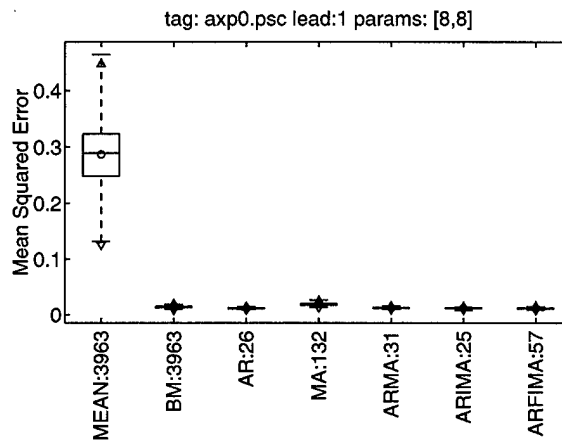
Figure 4.4: Axp0, 1 second lead, 8 parameters

parameters. MA models are clearly ineffective for load prediction.

4.4.2 Successful models have similar performance

Surprisingly, Figures 4.1 – 4.3 also show that the differences between the successful models are actually quite small. This is also the case if we expand to include testcases with 2 to 8 parameters instead of just 8 parameters. With longer lead times, the differences do slowly increase.

For more heavily loaded machines, the differences can be much more dramatic. For example, Figure 4.4 shows the distribution of one-step-ahead mean squared error measures for 8 parameter models on the axp0.psc trace. Here we see an expected raw variance (MEAN) of almost 0.3 reduced to about 0.02 by nearly all of the models. Furthermore, the mean squared errors for the different model classes are tightly clustered around the expected 0.02 value, quite unlike with MEAN, where we can see a broad range of values and the 97.5th percentile is almost 0.5. The axp0.psc trace and others are also quite amenable to prediction with long lead times. For example, Figures 4.5 and 4.6 show 15 and 30 second ahead predictions

tag: axp0.psc lead:15 params: [8,8]
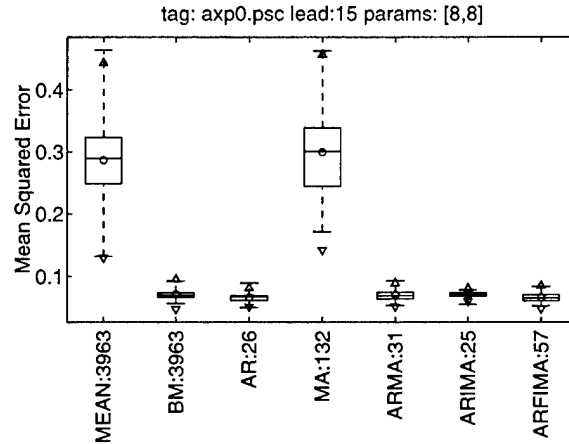
Figure 4.5: Axp0, 15 second lead, 8 parameters

tag: axp0.psc lead:30 params: [8,8]
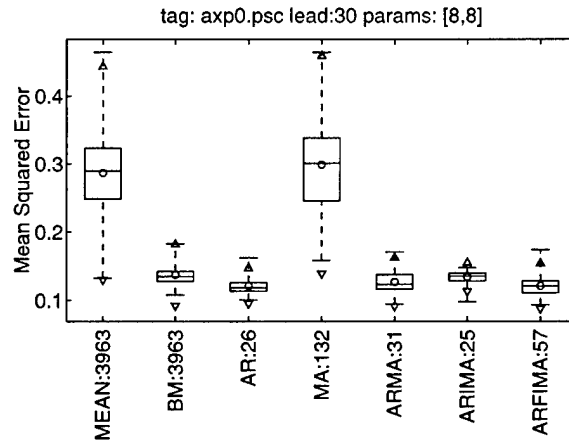
Figure 4.6: Axp0, 30 second lead, 8 parameters

for 8 parameter models on the axp0.psc trace. With the exception of the MA models, even 30 second ahead predictions are consistently much better than the raw signal variance. These figures remain essentially the same if we include testcases with 2 to 8 parameters instead of just 8 parameters.

Although the differences in performance between the successful models are somewhat small, they are generally statistically significant. We can algorithmically compare the expected mean squared error of the models using the unpaired t-test [66, pp. 209–212], and do ANOVA procedures to verify that the differences we detect are significantly above the noise floor. For each pair of model classes, the t-test tells us, with 95% confidence, whether the first model is better, the same, or worse than the second. We do the comparisons for the cross product of the models at a number of different lead times. We consider the traces both in aggregate and individually, and we use several different constraints on the number of parameters.

Table 4.2 shows the results of such a t-test comparison for the aggregated traces, a lead time of 1, and 8 parameters. In the figure, the row class is being compared to the column class. For example, at the intersection of the AR row and MA column, there is a '<', which indicates that, with 95% confidence, the

	MEAN	BM	AR	MA	ARMA	ARIMA	ARFIMA
MEAN	=	>	>	>	>	>	>
BM	<	=	=	<	=	=	=
AR	<	=	=	<	=	=	=
MA	<	>	>	=	>	=	>
ARMA	<	=	=	<	=	=	=
ARIMA	<	=	=	<	=	=	=
ARFIMA	<	=	=	<	=	=	=

Table 4.2: T-test comparisons - all traces aggregated, lead 1, 8 parameters. Longer leads are essentially the same except MA is always worse.

	MEAN	BM	AR	MA	ARMA	ARIMA	ARFIMA
MEAN	=	>	>	=	>	>	>
BM	<	=	>	<	=	=	>
AR	<	<	=	<	=	<	=
MA	=	>	>	=	>	>	>
ARMA	<	=	=	<	=	=	>
ARIMA	<	=	>	<	=	=	>
ARFIMA	<	<	=	<	<	<	=

Table 4.3: T-test comparisons - axp0, lead 16, 8 parameters.

expected mean squared error of the AR models is less than that of MA models, thus the AR models are better than MA models for this set of constraints. For longer lead times, we found that the results remained essentially the same, except that MA became consistently worse.

The message of Table 4.2 is that, for the typical trace and a sufficient number of parameters, there are essentially no differences in the predictive powers of the BM, AR, ARMA, ARIMA, and ARFIMA models, even with high lead times. Further, with the exception of the MA models, all of the models are better than the raw variance of the signal (MEAN model).

For machines with higher mean loads, there are more statistically significant differences between the models. For example, Table 4.3 shows a t-test comparison for the axp0.psc trace at a lead time of 16 seconds for 8 parameter models. Clearly, there is more differentiation here, and we also see that the ARFIMA models do particularly well, as we might expect given the self-similarity result of Section 3.5. However, notice that the AR models are doing about as well.

The results of these t-test comparisons can be summarized as follows: (1) Except for the MA models, the models we tested are significantly better (in a statistical sense) than the raw variance of the trace and the difference in expected performance is significant from a systems point of view. (2) The AR, ARMA, ARIMA, and ARFIMA models perform at least as well as the BM model, and perform better on more heavily loaded hosts. (3) The AR, ARMA, ARIMA, and ARFIMA models generally have similar expected performance.

4.4.3 AR models are better than BM and LAST models

Since the AR, ARMA, ARIMA, and ARFIMA models have similar performance when evaluated using the methods of the previous sections, the inclination is to prefer the AR models because of their much lower overhead (Chapter 2.5.5). However, should we prefer AR models over BM models, or over their degenerate case, the LAST model? After all, these latter models also have low overhead and are much

(a) AR(1) and BM(32)

(b) AR(2) and BM(32)

(c) AR(4) and BM(32)

(d) AR(8) and BM(32)

(e) AR(16) and BM(32)

(f) AR(32) and BM(32)
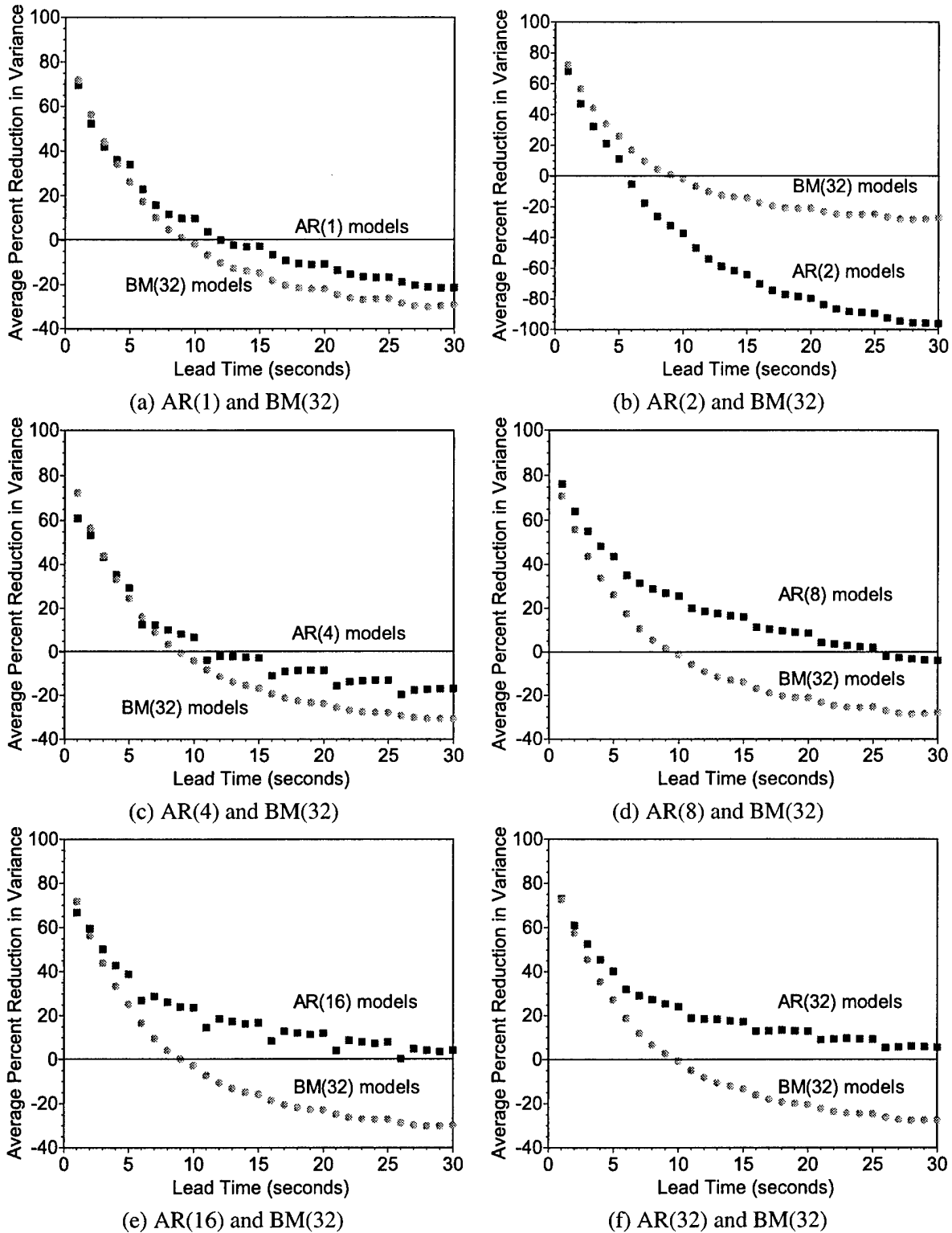
Figure 4.7: Paired comparisons of AR and BM models by lead time.

easier to understand and build.

To address this issue, we performed paired comparisons of testcases that used these models. Recall from
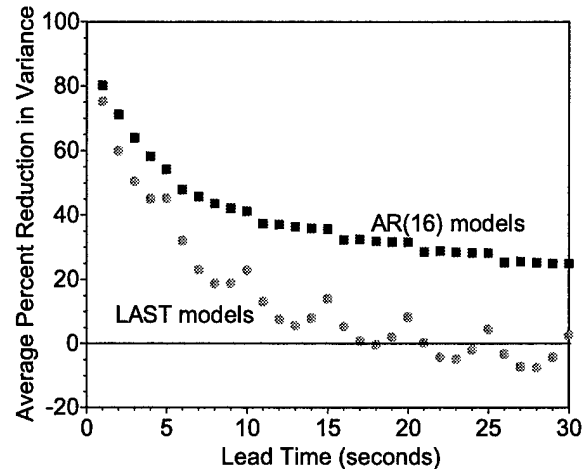
Figure 4.8: Paired comparisons of AR(16) and LAST models by lead time.

Section 4.3 that each testcase is run both with a randomly chosen model and with a BM model. For each testcase and lead time, we can directly compare the variance of the test interval (as measured by the MEAN model), the mean squared error of the randomly chosen model, and mean square error of the BM model. Further, we can normalize the reduction of variance each model provides at a particular lead time to the variance of the test interval itself (ie, (variance − mean squared error)/variance). Finally, we can aggregate these normalized reductions in variance across testcases.

Figure 4.7 uses this process to compare AR models of different orders to BM models. Each graph in the figure plots the average percentage reduction in variance as a function of the lead time for AR models of a given order and their corresponding BM models. The performance of the MEAN model is 0% in all cases. Each point on a graph encodes approximately 1000 testcases.

As the figure makes clear, AR models of sufficiently high order outperform BM models, especially at high lead times. Notice that for predictions as far ahead as 30 seconds, the AR(16) and AR(32) models provide lower variance than the raw load signal, while the BM models produce prediction errors that are actually more surprising than the signal itself. Indeed, the BM models only seem useful at lead times of 10 seconds or less. Where they shine as well as the AR models is at extremely short prediction horizons.

Figure 4.7 also shows that the performance of the AR models generally increases as their order increases. There are diminishing returns with higher order models, however. Notice that the gain from the AR(16) model to the AR(32) model is marginal. For very low order models, rather odd effects can occur, such as with AR(2) models (Figure 4.7(b)), which buck the overall trend of improving on lower order models. We noticed that AR models of order less than 5 have highly variable behavior, while AR models of order greater than 16 don't significantly improve performance.

Figure 4.7(a) shows the performance of an AR(1) model. If the only source of predictability of the host load signal was the exponential smoothing done in the kernel, we would expect that AR models of higher order would not provide better predictability. As we can see, however, higher order models do result in lower mean squared prediction errors.

It is interesting to compare AR(16) models to the simplest possible predictor, the LAST model. We ran an additional set of testcases comparing AR(16) and LAST models to do this. It is possible, of course, to mine the testcases in the original set to do the comparison. However, the testcases we would look at would not be random, but rather those where the BM model had selected LAST as being most appropriate. By running a separate set of testcases, we were able to eliminate this bias.

Figure 4.8 compares the average percentage reduction in variance, computed as before, of the AR(16)

model and the LAST model as a function of the lead time. As we can see, the AR(16) model significantly outperforms the LAST model, especially for lead times greater than five seconds. The AR(16) curve of Figure 4.8 is different from that of Figure 4.7(e) because of the use of a different random set of testcases.

4.4.4 AR(16) models or better are appropriate

Clearly, using one of the model classes, other than MA, for load prediction is beneficial. Further, using an AR, ARMA, ARIMA, or ARFIMA model is preferable to the BM model. In paired comparisons, AR models of sufficiently high order considerably outperform BM and LAST models, especially at longer lead times. Of this group of models, we found in Section 2.5.5 that the AR models are much less expensive to fit than the ARMA, ARIMA, or ARFIMA models and are of similar or lesser cost to use. In addition, AR models can be fitted in a deterministic amount of time. Because of this combination of high performance and low overhead, AR models are the most appropriate for host load prediction. AR models of order 5 or higher seem to work fine. However, we found the knee in the performance of the AR models to be around order 16. Since fitting AR models of this order, or even considerably higher, is quite fast, we recommend using AR(16)s or better for load prediction.

4.4.5 Prediction errors are not IID normal

As we described in Section 4.3, our evaluation of each testcase includes tests for the independence and normality of prediction errors. Intuitively, we want the errors to be independent so that they can be characterized with probability distribution function, and we want the distribution function to be normal to simplify computing confidence intervals from it.

For the most part, the errors we see with the different models are not independent or normal to any confidence level. However, from a practical point of view, the errors are much less correlated over time than the raw signal. For example, for AR(8) models, the Portmanteau Q statistic tends to be reduced by an order of magnitude or more, which suggests that those autocorrelations that are large enough to be significant are only marginally so. Furthermore, assuming normality for computing confidence intervals for high confidence levels, such as 95%, seems to be reasonable. Furthermore, if we want to predict the average host load over an interval of time, we will sum individual predictions. By the central limit theorem, the distribution of this sum should converge to normality, although it may do so slowly. In the next chapter, we successfully rely on this convergence.

4.5 Implementation of on-line host load prediction system

Using RPS, we were able to quickly implement an on-line host load prediction system based on the AR(16) model. Indeed, the initial implementation was simply to compose the prediction components described in Section 2.7. The predserver component can have its predictive model changed at run-time, and so it is trivial to configure it to use the AR(16) model. Later, we constructed a monolithic system with identical features. Both systems are described in Section 2.8, which also evaluates their performance and the added load they place on the host. To reiterate the conclusions, both the composed and monolithic systems have virtually unmeasurable overheads when used to predict the 1 Hz load signal with AR(16) models. Furthermore, they can be used at measurement rates 2-3 orders of magnitude higher than we require before saturating the CPU.

In the following chapters, we use the monolithic host load prediction system with the MEAN, LAST, and AR(16) models. This set of models lets us compare performance for the raw signal variance, an ad hoc predictor, and the formal predictor that we have found to be most appropriate. For each of these models, the system supplies predictions, mean squared error estimates, and the estimates of the covariances of the

prediction errors. In the case of the MEAN model, the mean squared error estimates are simply the measured variance of the signal, while the covariances correspond to the measured autocorrelation structure of the host load signal. For the LAST and AR(16) models, the mean squared error estimates and covariances are computed from the fitted model (LAST is treated as an AR(1)).

4.6 Conclusions

In this chapter, we applied the final two steps of the resource signal methodology of Section 2.1 to host load signals, specifically the 1 Hz Digital Unix five-second load average. The last chapter studied traces of such signals and concluded that linear models such as the Box-Jenkins models (AR, MA, ARMA, and ARIMA) might be appropriate for predicting them. However, it also found that host load was self-similar, which suggested that the more complex ARFIMA model might be needed. In addition, we found that host load exhibits epochal behavior, which could rule out linear models altogether.

To determine which, if any, of these models is truly appropriate, we studied their performance by running almost 200,000 randomized testcases on the August, 1997 set of load traces and then analyzing the results. We looked at predictions from 1 to 30 second into the future. In addition to the AR, MA, ARMA, ARIMA, and ARFIMA models, we also looked at the more intuitive BM and LAST models.

The main contribution of our evaluation is to show, in a rigorous manner, that host load on real systems is predictable to a very useful degree from past behavior by using linear time series techniques. In addition, we discovered that, while there are statistically significant differences between the different classes of models we studied, the marginal benefits of the more complex models do not warrant their much higher run-time costs. This is a somewhat surprising result given the complex properties we identified in the last chapter. Finally, we reached the conclusion that AR models of order 16 or higher are sufficient for predicting the host load signal up to 30 seconds in the future. These models work very well and are very inexpensive to fit to data and to use.

Using RPS, we implemented an on-line host load prediction system that uses the AR(16) model. This extremely low overhead system is described and evaluated in Section 2.8. In the next chapter, we will use the predictions of this system as the basis for predicting the running time of a task.

Chapter 5

Running Time Advisor

This dissertation argues for basing real-time scheduling advisors on explicit resource-oriented prediction, specifically on the prediction of resource signals. In the previous chapters, we presented a methodology and tools for understanding such signals and developing prediction systems for them. We then applied the approach to host load to develop an appropriate host load prediction system. In this chapter, we concentrate on how to use these resource-level predictions to provide application-level predictions, predictions of task running time.

When deciding on which host to run a task, applications and adaptation advisors such as the real-time scheduling advisor need to know the likely running time of the task on each of the available hosts. If they can modify the CPU demand of a task, perhaps by adjusting a quality parameter such as resolution, they may also be interested in the likely available processor time on each of the hosts. This chapter defines an interface for this purpose, describes an algorithm that uses host load predictions to implement the interface, and evaluates the performance of an implementation of that algorithm using a large variety of experimental environments. We refer to that implementation as the running time advisor.

To use the system, the application supplies the prospective host, the CPU demand of the task (expressed as the nominal running time on an unloaded host), and a confidence level. The system then collects the latest host load predictions and their estimated mean squared errors and correlations from the prospective host's load prediction system. It then uses the algorithm to compute an estimate of the running time of the task on that host.

The estimate the system reports contains two values. The first is the expected running time of the task on the host. The second is a confidence interval (as per the user's supplied confidence level) for the running time. For the purposes of the real-time scheduling advisor, we are most concerned about the confidence intervals that the system computes.

We evaluated the system using a real environment where the background load on a host is supplied by playing back a load trace. Using this technique, we can (re)construct a realistic repeatable workload on a real machine without limiting ourselves to synthetic workloads that may not capture the higher order behavior that host load prediction depends on, and which real load signals exhibit.

The host load predictions used to evaluate the system are provided by the on-line host load prediction system described in Chapter 2. The study described in Chapter 4 concluded that the most appropriate predictor to use for host load prediction was based on an AR(16) model or better. In evaluating the algorithm developed in this chapter, we use predictors based on the MEAN, LAST, and AR(16) models, which provide a measure of how well the algorithm works given (respectively) the raw variance of the load signal, the considerably lower mean squared errors of a simple predictor, and the mean squared error of the most appropriate predictor.

The evaluation consists of 114,000 randomized testcases run against each of our August, 1997 traces.

To understand the performance of the running time advisor, we data-mined these testcases.

We draw a considerable number of conclusions from our examination of the testcases. The main result is that our system, using a strong host load predictor such as the AR(16) predictor, indeed computes reasonable and useful confidence intervals for the running time of tasks. The expected running times that are computed are also quite accurate. The benefits of more sophisticated predictors depend on the how heavily loaded the host is and on the nominal time of the task. The 39 traces in the evaluation fit into five classes as far as the performance of our system is concerned. For most of these classes, and more than 90% of traces, the AR(16) predictor produces the best results.

5.1   Programming interface

When an application or application scheduler considers running a task on a particular host, it wants to know what the running time of the task on that host will be given the task's CPU demand. Alternatively, it may want to know the available CPU time during a given interval extending into the future.

The API we provide allows the application to ask these questions and get qualified answers. The running time advisor will use the host load predictions that the host makes available to compute the answers. The answers are qualified in the sense that they are expressed both as expected values and as confidence intervals. The confidence interval captures the uncertainty associated with prediction and modelling errors. The API has the following form:

```
int PredictRunningTime(Host                       &host,
                       RunningTimePredictionRequest  &req,
                       RunningTimePredictionResponse &resp);

struct RunningTimePredictionRequest {
  double conf;
  double tnom;
};

struct RunningTimePredictionResponse {
  double tnom;
  double conf;
  double texp;
  double tlb;
  double tub;
};

int PredictAvailableTime(Host                          &host,
                         AvailableTimePredictionRequest  &req,
                         AvailableTimePredictionResponse &resp);

struct AvailableTimePredictionRequest {
  double conf;
  double interval;
};

struct AvailableTimePredictionResponse {
  double conf;
  double interval;
  double texp;
  double tlb;
```

```
    double tub;
};
```

The first function, `PredictRunningTime`, is used to determine the running time of a task of nominal time tnom (t_{nom}) on the host. The request also includes a confidence level, `conf`, to describe how accurate the prediction needs to be. These components of the query are fields of the `RunningTimePredictionRequest` structure. The response returns in the `RunningTimePredictionResponse` structure. In addition to a copy of the request's fields, it includes the expected running time of the task, tnom (t_{exp}), as well as the upper and lower bounds of the `conf` confidence interval for the running time, [`tlb`, `tub`] ($[t_{lb}, t_{ub}]$). t_{exp} is a point estimate which represents the most likely running time. The actual running time, t_{act}, will likely be different from t_{exp} but be near it. The confidence interval represents a range of values around t_{exp} such that t_{act} will be in the range a fraction `conf` of the time.

The second function, `PredictAvailableTime`, is used to ask how much time is the available on the host during the next `interval` seconds. The request and response structures are similar to those already described. The differences are that the query contains the interval instead of the nominal task time, and that the response's estimates are of the available time.

5.2 Predicting running times

In this section, we describe the algorithm we developed for transforming from host load predictions and a task's nominal time to a prediction of the task's running time. We begin by describing the core algorithm, first illustrating how it works in continuous time, then discretizing it, and finally incorporating host load predictions. To compute confidence intervals using the algorithm, we must determine the variance of a sum of host load predictions, which turns out to be somewhat subtle because the prediction errors are themselves correlated. Next, we discover that the algorithm's predictions are erroneous because it does not model an important aspect of typical Unix schedulers. We introduce load discounting to repair the algorithm. Finally, we summarize the steps of the algorithm.

5.2.1 Core algorithm

In Chapter 3, we showed experimental results that related the running time of a task, t_{exec}, to the average load it experiences while it runs using the following continuous time model:

$$\frac{t_{exec}}{1 + \frac{1}{t_{exec}} \int_0^{t_{exec}} z(t)dt} = t_{nom}.$$

Here $z(t)$ is the load signal, shifted such that $z(0)$ is the value of the signal at the current time, t_{now}. We introduce this shift to simplify the presentation of our algorithm, and to conform to the Box-Jenkins notation for time series prediction that we used in the previous chapter. This simplification does not affect the results. t_{nom} is the nominal running time of the task, which quantifies the CPU demand of the task as its running time on an unloaded machine.

This continuous time equation is basically a fluid model of a priority-less host scheduler. We will use this simple model to describe our estimation procedure. However, real schedulers incorporate priorities that can change over time. We assume that the majority of the workload runs at similar priorities. In particular, we assume that there are no processes whose priorities have been drastically increased or decreased, such as with the Unix "nice" facility. Ultimately, we will relax this assumption slightly and model the temporary priority boosts that most Unix implementations give processes immediately after they become unblocked. Given this extension, the procedure we outline in this section works quite well.

Continuous time

The above equation is somewhat unwieldy to discretize and use, so, before we continue, let's define the *available time function*

$$at(t) = \frac{t}{1 + al(t)} , \ t > 0 \tag{5.1}$$

which depends on the *average load function*

$$al(t) = \frac{1}{t} \int_0^t z(\tau) d\tau , \ t > 0 . \tag{5.2}$$

$at(t)$ represents the available CPU time over the next t seconds, which is inversely related to the average load during that interval, $al(t)$. As the average load increases, the available time decreases. t_{exec} is then the minimum t for which $at(t) = t_{nom}$. Using this available time function makes it easier to explain how our algorithm estimates the running time of a task, and, of course, the available time function is offered directly through the API described in Section 5.1.

Discrete time

In our system, z is not a continuous-time signal, but rather it is a discrete-time $1/\Delta$ Hz sampling of that signal, $\langle z_{t+i} \rangle$, $z_{t+i} = z(i\Delta)$ for $i = 1, 2, \ldots, \infty$. Note the intentional similarity of this notation to that of the $1, 2, \ldots, \infty$-step-ahead predictions introduced in Chapter 3. We will indeed later replace these values with predictions. z_{t+1} represents $z(t)$ for $0 < t \leq \Delta$, and so on. We approximate $z(t)$ as $z(t) = z_{\lceil t/\Delta \rceil}$. This lets us write a discrete time approximation of $at(t)$ and $al(t)$:

$$at_i = \begin{cases} 0 & i = 0 \\ \frac{i\Delta}{1 + al_i} & i > 0 \end{cases} \tag{5.3}$$

$$al_i = \frac{1}{i} \sum_{j=1}^{i} z_{t+j} , \ i > 0 \tag{5.4}$$

at_i is the time available during the next $i\Delta$ seconds and al_i is the average load that will be encountered over the next $i\Delta$ seconds. We then estimate the available time $at(t)$ by linear interpolation:

$$at(t) = at_{\lfloor t/\Delta \rfloor} + \frac{t - \lfloor t/\Delta \rfloor}{\Delta} (at_{\lceil t/\Delta \rceil} - at_{\lfloor t/\Delta \rfloor}) \tag{5.5}$$

Host load predictions

Given these definitions, we substitute the predicted load signal $\hat{z}_{t+i}$ for z_{t+i}s resulting in the predicted average load $\hat{al}_i$, and then continue substituting back to give the predicted available time $\hat{at}_i$ and its corresponding continuous time approximation:

$$\hat{at}_i = \begin{cases} 0 & i = 0 \\ \frac{i\Delta}{1 + \hat{al}_i} & i > 0 \end{cases} \tag{5.6}$$

$$\hat{al}_i = \frac{1}{i} \sum_{j=1}^{i} \hat{z}_{t+j}, i > 0 \tag{5.7}$$

$$\hat{at}(t) = \hat{at}_{\lfloor t/\Delta \rfloor} + \frac{t - \lfloor t/\Delta \rfloor}{\Delta} (\hat{at}_{\lceil t/\Delta \rceil} - \hat{at}_{\lfloor t/\Delta \rfloor}) \tag{5.8}$$

Then, the expected running time of the task, t_{exp}, is simply the smallest t for which $\hat{at}(t) = t_{nom}$.

Confidence intervals

Because host load predictions are not perfect, we also report the running time or available time as a confidence interval, computed to a user-specified confidence level. The better the predictions are, the narrower the confidence interval is.

The predicted load signal is $\widehat{z}_{t+i} = z_{t+i} + a_{t+i}$, where z_{t+i} is the real value of the signal and a_{t+i} is the i-step-ahead prediction error term which we summarize with a variance $\sigma_{a,i}^2$. Our uncertainty in estimating the available time at_i is due to our uncertainty in estimating the average load al_i, which is due in turn to these error terms and their variance. To represent this uncertainty in the form of a confidence interval, we must push the underlying error variances through the equations above to arrive at a variance for the average load al_i.

Notice that the average load (Equation 5.7) sums the estimates $\widehat{z}_{t+i}$. Rewriting the equation, we can see that

$$\widehat{al}_i = al_i + \frac{1}{i}\sum_{j=1}^{i} a_{t+j} \tag{5.9}$$

By the central limit theorem, then, $\widehat{al}_i$ will become increasingly normally distributed with increasing i. Given that the errors a_{t+i} are of zero mean, $\widehat{al}_i$ has a mean (expected) value of al_i and a variance that depends on the sum of the prediction errors a_{t+i}:

$$\widehat{al}_i \sim N\left(al_i, Var\left\{\frac{1}{i}\sum_{j=1}^{i} a_{t+j}\right\}\right) \tag{5.10}$$

It is important to note that for short jobs or large Δ, this normality assumption may be invalid. We will evaluate the system later and determine whether the results of the assumption are reasonable.

Suppose the user requests a confidence interval at 95% confidence. We can then compute a confidence interval for al_i (for $i > 0$):

$$[al_i^{low}, al_i^{high}] = Min\left(0, \widehat{al}_i \mp \frac{1.96}{\sqrt{i}}\sqrt{Var\left\{\sum_{j=1}^{i} a_{t+j}\right\}}\right) \tag{5.11}$$

What this means is that we predict that al_i will be in the range $[al_i^{low}, al_i^{high}]$ with 95% probability. The 1.96 is the number of standard deviations of a standard normal needed to capture 42.5% of values. The Min is important because the average load cannot drop below zero, although the prediction errors can make that appear to be the case.

We can now back-substitute these upper and lower bounds of the confidence interval into $at(t)$ (Equation 5.5), resulting in upper and lower confidence intervals for $at(t) = [at^{low}(t), at^{high}(t)]$. Then the confidence interval on the running time is $[t_{lb}, t_{ub}]$, where t_{lb} is the minimum t for which $at^{high}(t) = t_{nom}$ and t_{ub} is the minimum t for which $at^{low}(t) = t_{nom}$.

5.2.2 Correlated prediction errors

Given the discussion of the previous section, we must still determine the variance of a sum of consecutive predictions in Equation 5.11 to compute the confidence interval. This is one of the subtler issues in converting from load predictions to running time predictions.

In essence, we are summing the one, two, i-step-ahead prediction errors, which are the random variables $a_{t+1}, a_{t+2}, \ldots, a_{t+i}$. Each of these random variables has a corresponding variance: $\sigma_{a,1}^2, \sigma_{a,2}^2, \ldots, \sigma_{a,i}^2$.

These variances are simply the mean squared error estimates produced by the predictor. We want to know the variance of their sum. By the central limit theorem, the distribution of such a sum converges to normality. It is tempting, but wrong, to compute the variance of the sum as

$$Var\left\{\sum_{j=1}^{i} a_{t+j}\right\} = \sum_{j=1}^{i} Var\{a_{t+j}\} = \sum_{j=1}^{i} \sigma_{a,j}^2 \qquad (5.12)$$

This would be valid if the prediction errors were independent, but it turns out that they are not. Almost by their very nature, linear models produce prediction errors which are correlated over time. The two-step-ahead prediction is not independent of the one-step-ahead prediction. Computing the variance of the sum in the above manner understates the variance, producing confidence intervals that are too small.

To see that the prediction errors are correlated, consider an AR(1) model, $z_{t+1} = \phi_0 z_t + a_{t+1}$. At time t, the remaining values will be:

$$
\begin{aligned}
z_{t+1} &= \phi_0 z_t + a_{t+1} \\
z_{t+2} &= \phi_0 z_{t+1} + a_{t+2} &= \phi_0^2 z_t + \phi_0 a_{t+1} + a_{t+2} \\
z_{t+3} &= \phi_0 z_{t+2} + a_{t+3} &= \phi_0^3 z_t + \phi_0^2 a_{t+1} + \phi_0 a_{t+2} + a_{t+3} \\
z_{t+n} &= \phi_0 z_{t+n-1} + a_{t+n} &= \phi_0^n z_t + \sum_{j=0}^{n-1} \phi_0^j a_{t+n-j}
\end{aligned}
$$

The predictions at time t must assume that all the white noise terms a_{t+k} are their expected values, which are zero, and thus

$$
\begin{aligned}
\hat{z}_{t+1} &= \phi_0 z_t \\
\hat{z}_{t+2} &= \phi_0 \hat{z}_{t+1} &= \phi_0^2 z_t \\
\hat{z}_{t+3} &= \phi_0 \hat{z}_{t+2} &= \phi_0^3 z_t \\
\hat{z}_{t+n} &= \phi_0 \hat{z}_{t+n-1} &= \phi_0^n z_t
\end{aligned}
$$

Now then, the one-step-ahead prediction error is a_{t+1}. The two-step-ahead prediction error is $\phi_0 a_{t+1} + a_{t+2}$, the value of which is clearly dependent on the one-step-ahead prediction error. The three-step-ahead prediction error is $\phi_0^2 a_{t+1} + \phi_0 a_{t+2} + a_{t+3}$, which depends on the one- and two-step-ahead prediction errors, and so on.

To correctly compute the variance of the sum of load predictions, we must compute the covariance of each of the prediction errors with each of the other prediction errors and then sum all i^2 terms of this covariance matrix. Entry j, k of this matrix is $CoVar\{a_{t+j}a_{t+k}\} = \sigma_{a,j}\sigma_{a,k}$ and is the covariance of the j-step-ahead prediction with the k-step-ahead prediction. Notice that variances of the individual predictions are simply the diagonal elements of the matrix.

The prediction errors' correlation over lead time is akin to a signal's autocorrelation over time. Recall that an autocorrelation sequence is simply a normalized autocovariance sequence. The covariances are easily computed from the autocovariance sequence.

In particular, $CoVar\{a_{t+j}, a_{t+k}\} = AutoCoVar\{a_t, a_{|j-k|}\}$.

The host load prediction system uses the algorithm of Box, et al to compute the autocovariance sequence for any linear model [23, pp. 159–160]. Since the LAST predictor is simply an AR(1) model with $\phi_0 = 1$, its autocovariances can also be computed using Box, et al's method. In the case of the MEAN predictor, the autocovariances are simply the autocovariances of the signal itself.

The client can request that the host load prediction system summarize the prediction errors either by the individual variances (the diagonal of the covariance matrix), the covariances of all the predictions (the entire covariance matrix), or the variance of the sum of the first $1, 2, \ldots, i$ predictions (the sum of the first i rows and columns of the covariance matrix). The variance of the sum of the first i predictions, which we will

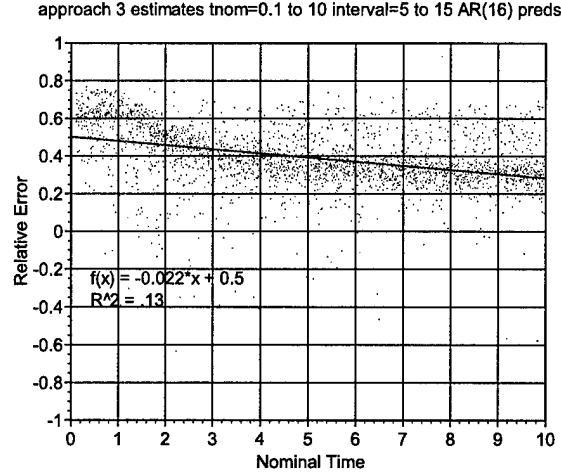approach 3 estimates tnom=0.1 to 10 interval=5 to 15 AR(16) preds

Figure 5.1: Relative errors of predictions as a function of nominal time without load discounting

write as $\sigma^2_{s,i}$ is then computed as

$$
\begin{aligned}
\sigma^2_{s,i} &= Var\left\{\sum_{j=1}^{i} a_{t+j}\right\} \\
&= SumVar\{\langle a_{t+1}, a_{t+2}, \ldots, a_{t+i}\rangle\} \\
&= \sum_{j=1}^{i}\sum_{k=1}^{i} CoVar\{a_{t+j}, a_{t+k}\} \\
&= \sum_{j=1}^{i}\sum_{k=1}^{i} \sigma_{a,j}\sigma_{a,k}
\end{aligned}
\tag{5.13}
$$

The sum is computed by the host load prediction system to avoid communicating the whole covariance matrix.

Typically, the non-diagonal elements of the covariance matrix are larger than zero. This results in Equation 5.13 being larger than Equation 5.12. This increased variance of the sum widens the confidence interval for al_i and, correspondingly, for the available time $at(t)$ and for the running time t_{exec}.

5.2.3 Load discounting

After we implemented the algorithm for computing the expected task running time (t_{exp}) of a task and its confidence interval ($[t_{lb}, t_{ub}]$) as described thus far in this section, we found that our results were somewhat dependent on the nominal running time of the task, t_{nom}. We will describe our evaluation methodology in detail in Section 5.3, but, for now, consider running, at a random time, a task with a randomly chosen nominal time t_{nom} on a host with an interesting background load. Before the task is run, we compute its expected running time, t_{exp}. Then we run the task and record its actual running time, t_{act}.

Figure 5.1 shows the result of running many such tasks. The figure plots the relative error (($t_{exp} - t_{act})/t_{exp}$) of the predictions versus the nominal time of the task. The figure plots 3000 tasks with nominal times chosen from a uniform distribution between 0.1 to 10 seconds. The interval between task arrivals was chosen from a uniform distribution between 5 to 15 seconds. The background load was the August, 1997 axp0 trace and the predictor used was an AR(16).

Notice that relative error is always positive and increases markedly as nominal time decreases. For one second tasks, the running time is over-predicted by as much as 80%. The confidence intervals are correspondingly skewed, with far too many points falling below the lower bounds of the intervals.

The problem is due to the Digital Unix scheduler, which is priority-based and which gives an "I/O boost" to processes (increases their priority) when a blocking I/O operation completes. For example, a process
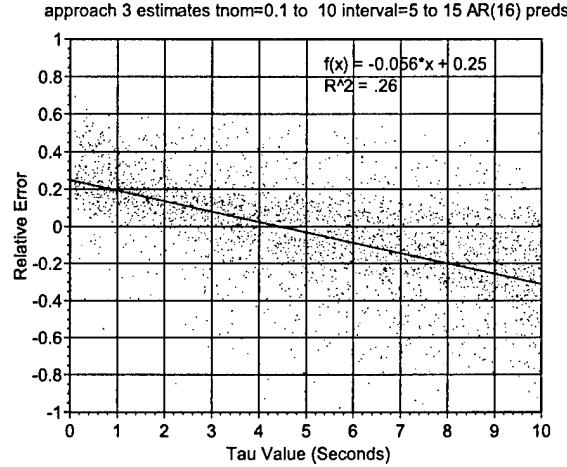
Figure 5.2: Relative errors of predictions as a function of $\tau_{discount}$, and derivation of $\tau_{discount}$ by linear fit.

(such as a CORBA ORB, an active frame server, or the spin server we use in our evaluation (Section 5.3.2)) that is blocked in a read on a network socket will have its priority boosted when the read completes. Over time, the process's priority will "age" to its baseline level. The result of this is that the awakened process will get more than its fair share of the CPU until its priority has degraded. The shorter the task, the more this mechanism will benefit it, and the more inaccurate our running time estimate will be, just as shown in Figure 5.1

Although we had originally intended to avoid modeling priority scheduling, leaving the modeling of a full priority-based scheduler for later, it was clear that we had to capture this priority boost effect.

Our solution is load discounting. Conceptually, when a task begins to run, its priority boost means that the background load on the system will effect it only slightly. As it continues to run, its priority drops and the background load becomes more and more important. We model this by exponentially decaying the load predictions $\widehat{z}_{t+j}$, the discounted load $\widehat{zd}_{t+j}$ being

$$\widehat{zd}_{t+j} = \left(1 - e^{-j\Delta/\tau_{discount}}\right)\widehat{z}_{t+j} \tag{5.14}$$

How quickly the initial load discounting decays depends on the setting of $\tau_{discount}$.

We determined the value of $\tau_{discount}$ empirically by again running a large number of randomized test-cases as described above. For this set of testcases, we used load discounting and chose $\tau_{discount}$ randomly from the range 0 to 10 seconds. Figure 5.2 plots the relative error of these testcases as a function of $\tau_{discount}$. Although there is plenty of dispersion (recall that these are point estimates t_{exp}, not confidence intervals) a linear trend clearly exists. We fitted a line to the points and determined that it crossed zero relative error at $\tau_{discount} = 4.5$ seconds.

Next, we ran a further large number of testcases with load discounting and $\tau_{discount} = 4.5$. The results are plotted in Figure 5.3. The figure plots the relative error as a function of the nominal time and is suitable for direct comparison with Figure 5.1. As can be seen, the appropriate $\tau_{discount}$ value has eliminated the dependence of the relative error on the nominal time and has further reduced the average relative error to almost exactly zero, which we would also expect from these point estimates.

Load discounting is an effective solution to the priority boosts that most Unix schedulers give to processes that have become unblocked. It is important to note, however, that other priority problems remain. For example, a background process which has had its priority significantly reduced (eg, a process which has has been "reniced") but which remains compute bound will result in artificially exaggerated predictions. Similarly, a background process with high priority will result in predictions that are too low.
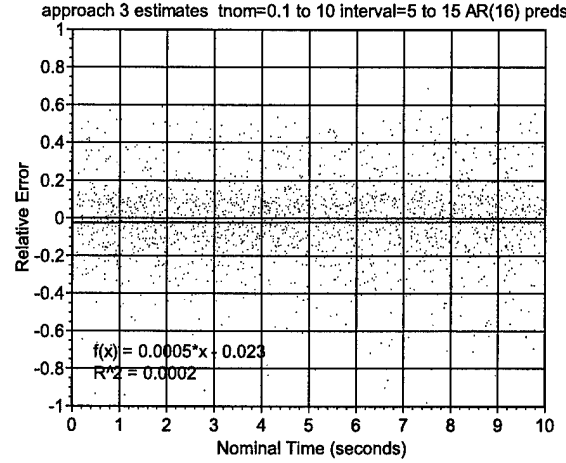
Figure 5.3: Relative errors of predictions as a function of nominal time with load discounting and appropriate $\tau_{discount}$ value.

5.2.4 Implementation

The following summarizes how the implementation implements the `PredictRunningTime` call.

1. Set a prediction horizon $n = 3\lceil t_{nom}/\Delta\rceil$.

2. Collect the latest predictions,
   $\widehat{z}_{t+1}, \widehat{z}_{t+2}, \ldots, \widehat{z}_{t+n}$, and the cumulative variances of their sums, $\sigma_{s,1}^2, \sigma_{s,2}^2, \ldots, \sigma_{s,n}^2$, from the host load prediction system running on the host. The variances are computed as per Equation 5.13.

3. Generate the discounted load predictions, $\widehat{zd}_{t+1}, \widehat{zd}_{t+2}, \ldots, \widehat{zd}_{t+n}$, as per Equation 5.14.

4. Using the discounted load predictions, compute the discrete time expected available time, $\widehat{at}_1, \widehat{at}_2, \ldots, \widehat{at}_n$ using the technique of Section 5.2.1, and the discrete time confidence intervals on the available time, $[at_1^{low}, at_1^{high}], [at_2^{low}, at_2^{high}], \ldots, [at_n^{low}, at_n^{high}]$ using the technique of Section 5.2.1.

5. If the lower bound on the maximum available time, at_n^{low}, is less than the nominal time, t_{nom}, increase the prediction horizon n and go to step 2.

6. To find the expected running time, t_{exp}, do binary search to find the smallest i for which $at_i > t_{nom}$, interpolate $at(t)$ around $i\Delta$ using Equation 5.8, and find t such that $at(t) = t_{now}$. This is t_{exp}.

7. Repeat the previous step with the at_i^{low} and at_i^{high} sequences to find the upper and lower bounds of the confidence interval, $[t_{lb}, t_{ub}]$.

The `PredictAvailableTime` call is implemented similarly.

5.3 Experimental infrastructure

The most appropriate way to evaluate the running time advisor is to actually run it on a host, submit tasks to that host, and measure how well the predicted times match the actual times. This section describes the infrastructure that we used to do this. We also used this infrastructure to evaluate the real-time scheduling advisor. The next chapter describes that evaluation.

The running time advisor is a composite whose errors are due to inaccurate host load predictions as well as modeling errors in transforming from the host load predictions and task CPU demand to task running time. We are interested in how this composite performs as a whole. The essential comparison to make when evaluating the system is the comparison between predicted and actual running times. To make such a comparison in a simulation would require a method for realistically estimating the actual running time, but that is a part of the system we want to evaluate! The measurement-based approach avoids this problem because the actual running time is measured directly.

The goal of our experimental infrastructure is to provide a realistic, repeatable environment for evaluating the running time advisor, and, in the next chapter, the real-time scheduling advisor). The infrastructure hardware consists of two Alphastation 255 hosts connected with a private network. Both machines run Digital Unix 4.0D. One host is referred to as the measurement host while the other is called the recording host. The hosts have no other load on them.

The recording host runs software that interrogates the components running on the measurement host and submits tasks to it. The measurement host runs the following components:

- A load playback tool operating on some load trace

- A host load sensor

- One or more host load prediction systems

- A spin server

The load playback tool, described in the next section, performs work that replicates the workload captured in the load trace. The choice of load trace is experiment-specific. The host load sensor provides an interface for the recording host to request the latest load measurement on the host. The host load prediction systems are described in Section 2.8.1 and provide an interface for the recording host to request the latest load predictions using some experiment-specific prediction model. The spin server runs tasks—it takes requests to compute (using a busy loop) for some number of CPU-seconds and then returns the wall-clock time that the task took to complete.

The remainder of this section describes the load playback tool and the spin server in more detail.

5.3.1 Load trace playback

To evaluate the running time advisor, a realistic background workload is necessary. The load signal produced by running this workload should have a realistic correlation structure, since it is precisely this that a predictor attempts to model.

To meet these requirements, our infrastructure uses *load trace playback*, a new technique in which a workload is generated according to a load trace. With no other work on the host, this background load results in the host's load signal repeating that of the load trace. Furthermore, we can repeatedly play back the same load and explore the prediction system's behavior using the traces from many different hosts. Because the workload reflects the trace, which certainly records the real behavior of a host, it is suitable for evaluating prediction-based systems.

Playback may seem like overkill, but it is not. As we discovered in Chapter 3, host load signals typically have complex behaviors. They typically have long, complex autocorrelation structures which can change abruptly. Our prediction systems exploit the autocorrelation structure to increase predictability and treat abrupt changes as points at which models need to be refit. Any synthetic load generator would have to reproduce these properties, and that is difficult. Furthermore, there may be other properties of the signals that we do not yet understand, but which may be important for prediction. For these reasons, playing back actual load behavior is most appropriate.

(a) Target trace

(c) Error histogram

(b) Measured trace

(d) Error autocorrelation
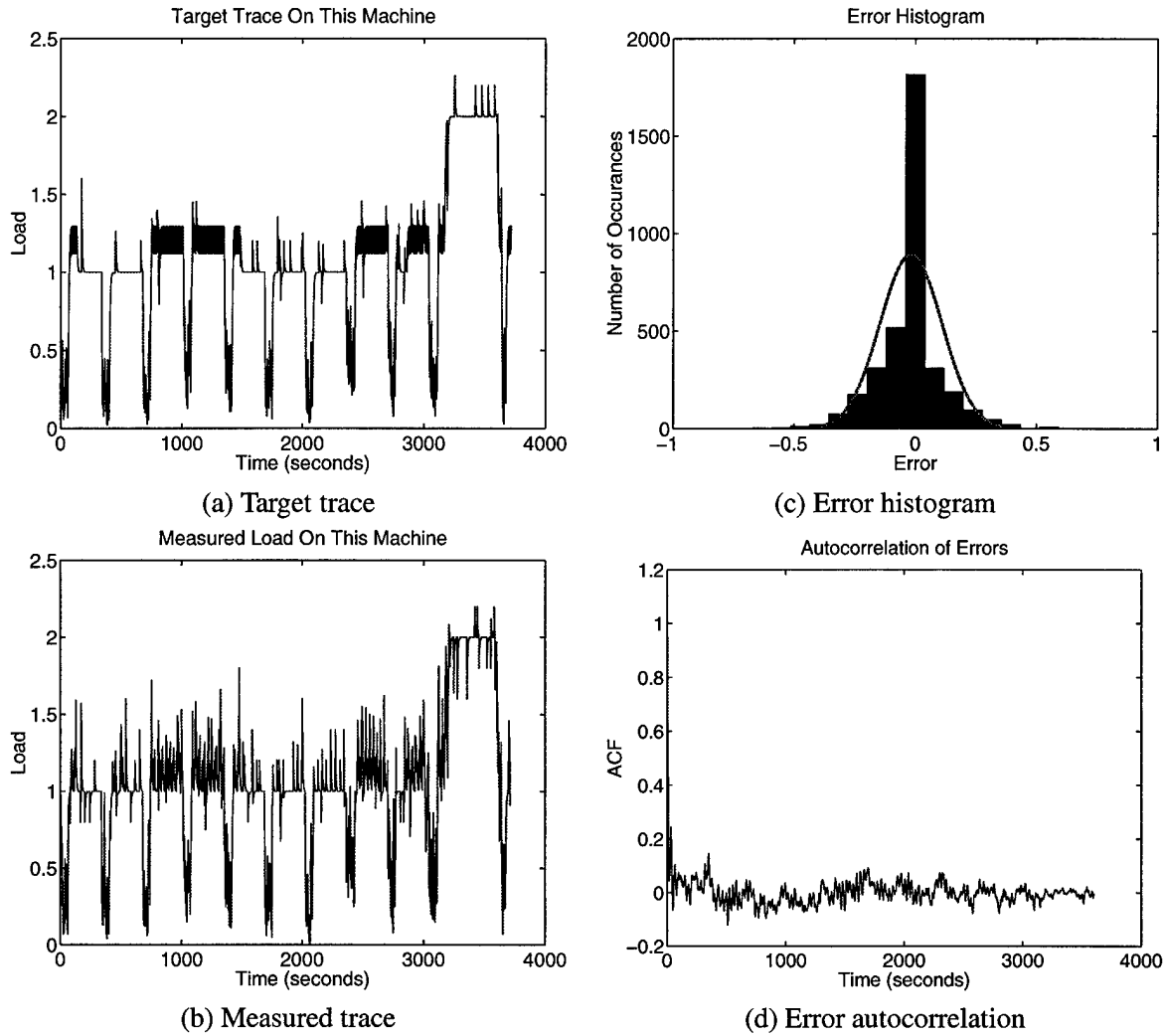
Figure 5.4: Example of host load trace playback.

Figure 5.4 is an example of load trace playback. In this case, the trace collected on the host axp0.psc on August 23, 1997 is being played back. Figure 5.4 plots (a) the target load signal, (b) the measured load signal, (c) the histogram of playback errors, and (d) the autocorrelation function of the playback error for playback of the first hour of the trace. As can be seen, the measured load tracks the target load with low and largely uncorrelated error. There is a slight cumulative delay in the playback process because work-based playback (explained later) is being used. Another extraneous source of playback error is the modicum of other work on the machine. The histogram shows us that most of the errors are quite small. The histogram also slightly overstates the actual error in terms of the work performed. This is due to other extraneous load on the system, sampling issues, and the effects of the kernel's smoothing filter, which we discuss later in this section. More details on host load playback, including its performance on platforms other than DUX, can be found elsewhere [33].

The playback algorithm

To describe the operation of the system, we'll focus on a periodically sampled trace. The load playback tool can also operate with nonperiodically sampled traces.

The load average is an exponential average of the number of processes in the system's ready-to-run queue. Conceptually, the length of the ready queue is periodically sampled, these samples x_i flow through an exponential filter

$$z_{i+1} = (e^{-\Delta/\tau_{record}})z_i + (1 - e^{-\Delta/\tau_{record}})x_i \tag{5.15}$$

where Δ is the sampling interval and the application-visible load average is the z_i series. A load trace is gathered by periodically sampling the load average and recording the time-stamped samples to a file. The sample rate should be at least twice is high as the sample rate of the in-kernel process, so we rewrite the above equation like this:

$$trace_{i+1} = (e^{-\Delta/2\tau_{record}})trace_i + (1 - e^{-\Delta/2\tau_{record}})x_i' \tag{5.16}$$

In load trace playback, we want the measured load average to track the load average samples in the trace file. To do this, we treat the x_i' in the above equation is the expected run-queue length during the last $\Delta/2$ seconds. To determine the x_i', we "deconvolve out" the smoothing function using the method described by Arndt, et al [7]. All that is necessary to do this is knowledge of the smoothing constant τ_{record} for the machine on which the trace was taken. For Digital Unix, τ_{record} is 5 seconds. On most other Unix systems, τ_{record} is 60 seconds. A larger τ_{record} value indicates that the load average will behave more smoothly.

A value x_i' represents the amount of contention the CPU saw for the time interval. To reproduce this contention, we split the interval into smaller subintervals, each of which is larger than the scheduler quantum, and then stochastically assign subprocesses to either work or sleep during the subintervals. For example, suppose $x_i' = 1.5$. We would assign subprocess 0 to work during a subinterval with probability 1, and subprocess 1 to work during a subinterval with probability 0.5.

After each individual x_i' value has been played, the load average on the system is measured. This measurement, $measure_i$, is compared against the load trace *as it would have appeared given the* $\tau_{playback}$ *of the playback machine*, $trace_i'$. $trace_i'$ is determined by smoothing the x_i' series with an exponential averaging filter having the playback machine's $\tau_{playback}$.

Interacting with other load

Most likely, there will be other processes producing load on the system—the tasks we will run during the evaluation, for example. It is important to understand how the load introduced by the load playback tool reacts to this other load. Two modes of operation are possible, time-based and work-based. In our evaluation we use work-based playback.

In the time-based mode of operation, the contention probabilities implied by the x_i' value last only till the end of x_i''s interval in real time. Essentially, this means that the other load on the machine can only amplitude modulate the played back load. For example, suppose there is a uniform 1.0 load on the machine and the load trace dictates a 1.0 load from 0 to 1 second and zero elsewhere. Then, the measured load will (ideally) be 2.0 from 0 to 1 second and 1.0 elsewhere.

In the work-based mode of operation, the load is interpreted as work that must always be done, but which may be slowed down by other load. This means that the other load on the system can also frequency modulate the played load. Using the same example as the previous paragraph, the measured load would be 2.0 from 0 to 2 seconds and 1.0 elsewhere.

The load playback tool also supports a negative feedback mechanism. Using this mechanism, it can sometimes more accurately track the load average in the trace. However, the feedback mechanism will also

try to correct for other load on the system, trying to make the total load track the trace load. Because of this obvious flaw, we do not use it in our evaluation.

Imperfections

The measured load during playback tracks the desired load only approximately in some cases. This is mainly because because the OS's scheduler is functioning both as a sampling processes and as part of the sampled process.

The sampling process that produces the x_is in the Equation 5.15 is the OS's scheduler, which sees only integral numbers of processes on the ready queue. We treat our approximation of x_i, x_i', as the expected value of x_i, $E[x_i]$. In playing back x_i' we contrive to make the value the scheduler samples this expected value. However, the second moment, $E[x_i^2]$ is nonzero, so the actual value sampled may be different even if the playback is dead on. Consider the following contrived example. Suppose $x_i' = 0.5$, we have a subprocess spend 50% of its time sleeping and 50% working, and we alternate randomly between these extremes. The expected run queue length the scheduler would see is then $E[x_i] = (0.5)(1) + (0.5)(0) = 0.5$. However, the scheduler will really sample either 1 or 0. The effect on the load average in either case will be drastically different than the expected value resulting in error. Furthermore, because the load is an exponential average, that error will persist over some amount of time (it will decline to 33% after τ_{record} seconds). Another way of thinking about this is to consider the second moment, $E[x_i^2]$. For this example, $E[x_i^2] = (0.5)(1)^2 + (0.5)(0)^2 = 0.5$, so the standard deviation of the distribution of $E[x_i]$ is $\sqrt{E[x_i^2] - E[x_i]^2} = \sqrt{0.5 - 0.25} = 0.5$ which explains the variability in what the scheduler will actually sample.

Even if the sample rate of the trace is low compared to the sample rate of the scheduler, resulting in the x_i''s corresponding measurements being derived from more than one observation by the scheduler, any extant error is still propagated by the exponential filter.

Another source of error is that an actual process on the ready queue may not consume its full quantum.

Real traces versus synthetic traces

In almost all cases, when a real load trace that has been sampled at a sufficiently high rate is played, the x_i's are close to integers. Intuitively, this makes sense - the scheduler can only observe integral numbers of processes on the ready queue, and if our estimates of its observations (the x_i's) are accurate, they should also mostly be integers.

It's easy to construct synthetic traces that are actually not sensible in terms of short term behavior. One such trace is the continuous 0.5 example discussed in the previous section. Deconvolving the trace produces x_i's slightly above 0.5. Load playback reasonably produces a situation where the expected ready queue length is 0.5 by having a process spend half of its time working and half sleeping. However, the observations the kernel (the x_is) will make will be either 0 or 1. Thus the measured load trace will vary widely. The average load average will match the 0.5 we desire, but the short term behavior will not. It's important to note that load playback tool is doing what the user probably wants (keeping the CPU 50% busy), but that the load average fails to be a good measure for how well the generated load conforms.

Another way a synthetic trace can fail is to have very abrupt changes. For example, a square wave will be reproduced with lots of overshoot and error. In order for the abrupt swings of a square wave to have been the output of the exponential smoothing, the input x_i must have been much more abrupt, and the estimate x_i' will also be quite large. This means load playback has to have many processes try to contend for the CPU at once, which raises the variability considerably.

The best way to produce synthetic traces is to create a string of integer valued x_is and smooth them with

the appropriate τ_{record}. Another possibility is to present these x_is directly to load playback with a very small τ_{record} trace value.

5.3.2 Spin server

The spin server runs submitted tasks and measures how long they take to complete. A task consists of the number of CPU seconds that should be run. The spin server uses a busy loop that periodically checks the accounted system and user time (using the getrusage system call) that the process has consumed. The loop exits when the requested amount of CPU seconds have been consumed and reports the wall clock time used. At startup, the spin server calibrates itself with respect to the host's running rate and the overhead of the getrusage call. This permits the loop to be extremely precise about the amount of computation it does using a minimal number of getrusage calls. The relative absolute error is much less than 1%.

5.4 Evaluation

To evaluate the running time advisor, we ran 114,000 randomized testcases using the infrastructure and studied the results. The testcases were randomized with respect to their starting time, their nominal running time (0.1 to 10 seconds), the underlying host load predictor that was used (MEAN, LAST, and AR(16)), and the load trace used to generate the background load (all of the August, 1997 traces). We characterized the quality of the expected running times and confidence intervals produced by the system using three metrics. We evaluated the effect on these metrics of the different traces, load predictors, and nominal running time.

The main conclusion is that the running time advisor does indeed produce high quality predictions for task running times. Performance depends most strongly on the choice of host load predictor. There is a marked increase in performance in moving from the MEAN predictor (where prediction errors are due the raw signal variance) to the LAST predictor (which is the simplest predictor to take advantage of the autocorrelation of load signals). There is a smaller, but important, gain in moving from the LAST predictor to the AR(16) predictor (which is the predictor that we found most appropriate for host load prediction in the previous chapter). The nature of these gains depends on how heavily loaded the host is. Performance also depends on the running time of the task and generally improves as running time increases.

5.4.1 Methodology

To evaluate the running time advisor given a particular traced host, we start up the experimental infrastructure described in Section 5.3 on the measurement and recording hosts. The load playback tool is set to replay the selected trace. The host load sensor is configured to run at 1 Hz. Three host load prediction systems are started: MEAN, LAST, and AR(16). The systems are configured to fit to 300 measurements (5 minutes) and to refit themselves when the absolute error for a one-step-ahead prediction exceeds 0.01 or the measured one-step-ahead mean squared error exceeds the estimated one-step-ahead mean squared error by more than 5%. The minimum interval between refits is 30 seconds and maximum interval before the measured mean squared error is tested is 300 seconds.

The prediction and measurement software are permitted to quiesce for at least 600 seconds. Then 3000 consecutive testcases are run on the recording host, each according to this procedure:

1. Wait for a delay interval, $t_{interval}$, selected from a uniform distribution from 5 to 15 seconds.

2. Get the current time t_{now}.

3. Select the task's nominal time, t_{nom}, randomly selected from a uniform distribution from 100 ms to 10 seconds.

4. Select a random host load prediction system from among MEAN, LAST, AR(16).

5. Use the `PredictRunningTime` API to compute the expected running time t_{exp} and the 95% confidence interval $[t_{lb}, t_{ub}]$ using the latest available predictions available from the selected host load prediction system.

6. Run the task on the spin server and measure its actual running time, t_{act}.

7. Record the timestamp t_{now}, the prediction system used, the nominal time t_{nom}, the expected running time t_{exp}, the confidence interval $[t_{lb}, t_{ub}]$.

After all 3000 testcases have been run, their records are imported into a database table corresponding to the trace.

It takes approximately 13 hours to complete 3000 testcases. To evaluate the running time advisor, we then mined the database.

5.4.2 Metrics

The `PredictRunningTime` function predicts the running time of a task in two ways: as an expected value and as a confidence interval. Because the lower bound of the confidence interval is artificially limited due to the fact that load cannot drop below zero, the expected time is not necessarily in the middle of the confidence interval.

Evaluating the quality of the confidence interval, $[t_{lb}, t_{ub}]$, is a somewhat complex endeavor. Suppose we generate run a wide variety of testcases with a specified confidence, say 95%. If we used the ideal algorithm for computing confidence intervals and the best possible predictor, the lengths of the tasks' confidence intervals would be the minimum possible such that 95% of the tasks would have running times in their predicted intervals. An imperfect algorithm, such as ours, will compute confidence intervals that were larger or smaller than ideal where more or fewer than 95% of the tasks complete in their intervals. The important point is that to evaluate a confidence interval algorithm, we must measure the lengths of the confidence intervals it produces, and the number of tasks which complete within these confidence intervals. We used following two metrics:

- **Coverage**: the fraction of tasks which complete with their predicted confidence intervals

- **Span** : the average width of the confidence interval width in seconds

The ideal system will have the minimum possible span such that the coverage is 95%.

To evaluate the quality of the expected running time, t_{exp}, we are interested in how strongly the actual time, t_{act}, correlates with it. Imagine plotting the actual times versus the expected times for all the tasks. With an perfect predictor, the result would be a perfect 45 degree line starting at 0. An imperfect predictor, such as ours, will approximate the line but with points scattered around it. The degree of this scatter is measured by the R^2 value of the linear fit. $R^2 = 1$ would correspond to a perfect predictor while $R^2 = 0$ corresponds to randomness. We will evaluate the quality of the expected times using this R^2 value, which we will simply refer to as the $\mathbf{R^2}$ value. This is not the entire story. We also care about the slope and intercept of the line. We discuss this further in Section 5.4.3.

5.4.3 Results

Running 3000 randomized testcases on each of the 39 traces in our August, 1997 set of traces resulted in 114,000 testcases to mine. This plethora of testcases allows us to characterize the performance of the running

time advisor in a wide variety of ways. Furthermore, because the testcases are randomized, our results generalize to hosts for which our set of traces are representative, which we believe to be a considerable portion of hosts.

We want to answer several questions. The most important of these is whether our system does indeed provide useful predictions of task running times, both in terms of the expected running time and in terms of the confidence interval for the running time. We are most interested in the confidence intervals because these are the predictions that we will use in the next chapter to make real-time scheduling decisions. In addition we want to understand how the choice of underlying host load predictor affects prediction performance, and how that performance depends on the nominal time of the task.

To address these questions, we looked at the testcases in five ways. First, we measured the quality of the confidence intervals independently of the nominal time of the task. For each trace, we computed the confidence interval metrics of coverage and span. Then we compared the different predictors based on these per-trace metrics. Next, we conditioned this comparison on the nominal time of the task, dividing the range in to small, medium, and large tasks. The third and fourth steps repeat this approach but using the R^2 metric for the expected running time. Finally, we hand-classified each trace based on the relationship of the performance metrics and the nominal time. This resulted in five classes. We then developed a recommendation for each class.

The overall conclusion is that the system does indeed predict reasonable point estimates and confidence intervals for task running times. Using the AR(16) predictor, we found that there were only five traces (out of 39) in which fewer than 90% of the tasks completed in their confidence interval, and only one host where fewer than 90% were within their computed confidence intervals. The relationships between MEAN, LAST, and AR(16) depend on whether the host is heavily loaded or not. On hosts with high load, the more sophisticated predictors were able to produce significantly better coverage by estimating wider spans. On hosts with low load, they achieved appropriate coverage with much smaller spans. The AR(16) predictor generally produced better results than LAST, and much better than MEAN. Performance generally improved as nominal time was increased. In terms of predicting the expected running time, although there were significant differences between the predictors, none of them could be considered to have the "best" performance. Interestingly, unlike with confidence intervals, the expected running time predictions grew worse with increasing nominal time.

Nominal time independent evaluation of confidence interval quality

Consider mapping a task with a nominal time of 0.1 to 10 seconds on a randomly selected host. Will the `PredictRunningTime` function produce an accurate confidence interval for its running time? How will the accuracy depend on which underlying host load predictor is used?

Figures 5.5, 5.6, and 5.7 provide answers to these questions. These figures present the same information, but arranged in different forms to more clearly illustrate different observations. For example, Figure 5.5 plots the (a) coverage, and (b) span for each of the load traces in our study. Consider Figure 5.5(a). The coverage of trace axp1 using the MEAN predictor is approximately 0.62. This point is the result of running 3000 testcases (Section 5.4.1) on the axp1 trace, selecting the approximately 1000 of these that used the MEAN predictor, counting the number of those testcases in which the deadline was met, and dividing by the total number of MEAN testcases. Recall that our goal is a 95% confidence interval, so MEAN is clearly inappropriate. We can see, however, that for the same trace, the AR(16) predictor provides much better coverage, about 92%, while the LAST predictor does slightly better at about 98%. If we look at the axp1's span metric in Figure 5.5(b), we can begin to understand why. To produce the point for the MEAN metric, the confidence interval widths for the approximately 1000 MEAN testcases were averaged. We see that the span is about 3.4 seconds. The AR(16) predictor's span, computed similarly, is a slightly higher (worse) 4
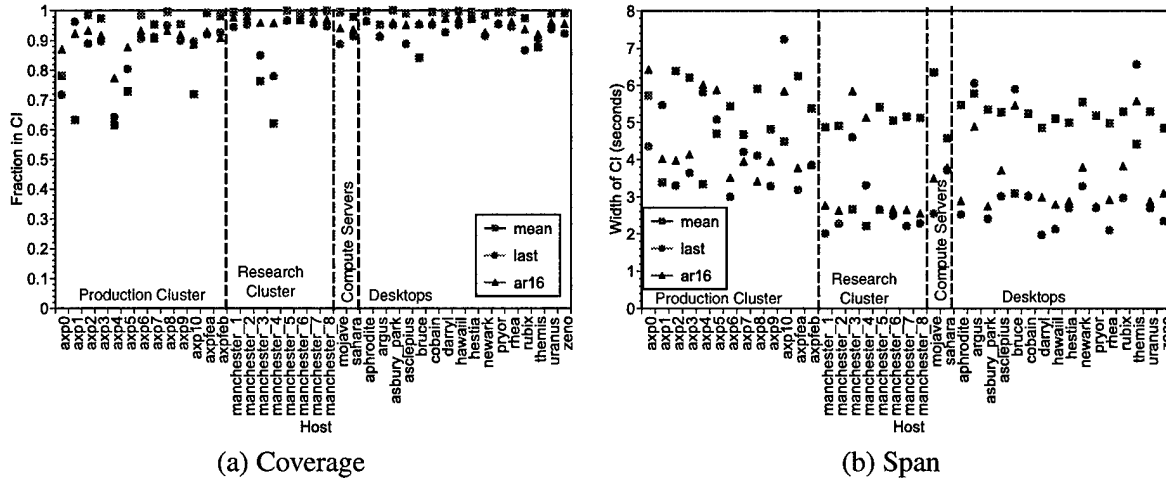
(a) Coverage   (b) Span

Figure 5.5: Confidence interval metrics for 0.1 to 10 second tasks on all hosts, independent of nominal time.

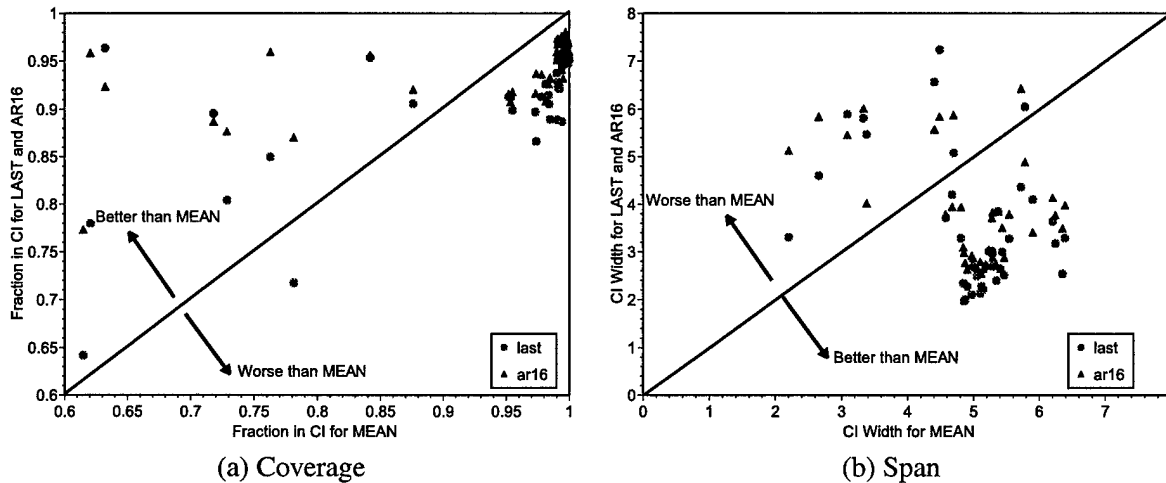(a) Coverage   (b) Span

Figure 5.6: Confidence interval metrics showing LAST and AR(16) versus MEAN for 0.1 to 10 second tasks on all hosts, independent of nominal time.

seconds, while the LAST predictor's span is a huge 6.4 seconds. The small increase in span going from the MEAN to the AR(16) predictor brings the coverage from 0.62 to 0.92, which is very close to the target of 0.95. The much greater increase in span going from AR(16) to LAST overshoots the target coverage, going as far as 0.98.

As can be seen from Figure 5.5(a), for almost every trace in our study, there is some predictor that provides a coverage that is near our target 95%. This is an important result that supports our claim of being able to compute accurate confidence intervals. However, perhaps the predictors are producing inordinately wide spans, reducing their usefulness. The spans in Figure 5.5(b) are difficult to interpret. There is clearly a *difference* between the predictors, and the span of the MEAN predictor seems to usually be much wider than that of the LAST and AR(16) predictors. However, there are some traces where the more sophisticated predictors seem to run aground, resulting in larger spans than MEAN.

Figure 5.5 is powerful for making observations about individual hosts, but it is somewhat difficult to compare predictor performance between hosts using it. Figure 5.6 rearranges the data to make such comparisons easier. We plot the performance metrics' values for LAST and AR(16) for each trace versus the

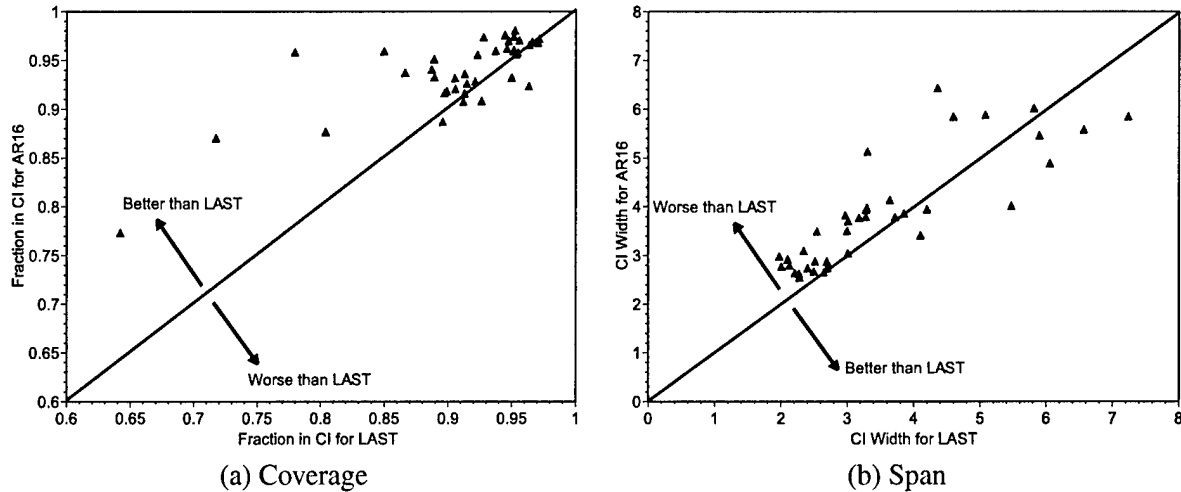(a) Coverage   (b) Span

Figure 5.7: Confidence interval metrics showing AR(16) versus LAST for 0.1 to 10 second tasks on all hosts, independent of nominal time.

metrics values for the MEAN predictor. This helps to answer the question: What is the benefit of using a more sophisticated predictor than MEAN? For example, consider Figure 5.6(a). For each trace, a dot is plotted at (coverage of MEAN testcases, coverage of LAST testcases), and a triangle is plotted at (coverage of MEAN testcases, coverage of AR(16) testcases). On the graph, a 45 degree line separates the regions where the coverage is better than MEAN from that where it is worse. Figure 5.6(b) is arrived at with exactly the same methodology, except using the span metric.

Figure 5.6(a) and (b) make it clear what the LAST and AR(16) predictors generally provide quite different performance results than the simple MEAN predictor. For nine of the traces, the more sophisticated predictors provide significantly better coverage than the MEAN predictor. For the remainder of the traces, the more sophisticated predictors have slightly lower coverage. In terms of the span metric, nine of the traces show significantly wider spans than MEAN, while the remainder are much narrower.

The nine traces on which the LAST and AR(16) produce better coverage performance are the same traces in which they produce worse span performance than MEAN. These nine traces correspond to the hosts that exhibit greater mean load (and correspondingly, greater variability in load (Chapter 3)). The LAST and AR(16) predictors are better able to "understand" such hosts and compute appropriately wider confidence intervals compared to MEAN. These wider confidence intervals result in a far greater chance of a task's actual running time falling within its computed confidence interval. This is precisely the behavior that we want. Our goal is that 95% of tasks fall within their confidence intervals. With the AR(16) predictor, only 5 cases are less than 90% and only one less that 85%, whereas with the MEAN predictor, only one of the high load traces is better than 85%. The gain from MEAN to AR(16) can be as much as 30%, and it is typically around 15%.

Two effects are at work here. First, the predictions of the LAST and AR(16) predictors depend most strongly on recent measurements. The MEAN predictor, on the other hand, always presents the long term mean of the signal. As a result, the LAST and AR(16) predictors will respond much more quickly during the period after an epoch transition (Chapter 3) before a model refit happens. This means that their predictions, and thus the center point of the confidence interval will much more likely be in the right place. The second effect results from how the confidence interval length is computed. Recall that with the MEAN predictor the autocovariance of the signal is used to compute the confidence interval, while for the LAST and AR(16) predictors it is the autocovariance of their prediction errors that is used. On a high load, high variability

host, an epoch transition is more likely than on a low load, low variability host to make the autocovariance of the signal fail to characterize the new epoch well. In contrast, the structure of the autocovariance of the prediction errors will probably change considerably less drastically.

For those hosts which have lower load and variability, the LAST and AR(16) predictors produce significantly narrower confidence intervals than MEAN while still capturing a reasonable number of tasks within their computed confidence intervals. On average, the confidence intervals are shrunk by 2-3 seconds while the fraction of tasks within their confidence intervals shrinks by about 5%. Since for these lightly loaded hosts, the MEAN predictor results in coverages that are larger than the target 95%, this is not an unreasonable tradeoff. Essentially, for these low load hosts, moving from MEAN to AR(16) reduces coverage by about 5% while decreasing the span by 2-3 seconds (about 33%).

At this point, we have shown that our algorithm does indeed compute reasonable confidence intervals for task running times and that it does so more accurately when using a more sophisticated predictor than MEAN. Now we would like to know whether we should prefer the LAST predictor or the AR(16) predictor. We have already pointed out some of the differences between these two. To continue, it is useful to plot the data differently once again. Figure 5.7 plots the performance of the AR(16) predictor versus the LAST predictor. From Figure 5.7(a), we can see that the confidence intervals computed using AR(16) generally include more of their tasks than those computed using LAST. Using the AR(16) predictor, only four of the cases are at less than 90% and only one less than 85%. Using LAST, 9 are less than 90%, while four are less than 85%. This gain is due to AR(16) predictors producing slightly wider confidence intervals, as can be seen from Figure 5.7(b). The span increases by 0.5 to 1 second in moving from LAST to AR(16).

On heavily loaded hosts, the gain in moving from LAST to AR(16) is more significant—either we see a large increase in coverage, on the order of 10% or more, or there is a slight decline of 5% or significantly less. Correspondingly, the span either grows on the order of 2-3 seconds or it shrinks by 1-2 seconds.

To summarize, the benefit of using the LAST or AR(16) predictor over MEAN is two fold. For heavily loaded hosts, the more sophisticated predictors compute wider confidence intervals which lead to a much larger fraction of the tasks running within their confidence intervals. For lightly loaded hosts, LAST and AR(16) produce much narrower confidence intervals than MEAN at a small, reasonable cost to coverage.

The benefit of using the AR(16) predictor over the LAST predictor is also two fold. For heavily loaded hosts, on about half of these traces it generally produces wider confidence intervals that significantly increase coverage, driving it much closer to the target of 95%. On the other half it, shrinks their confidence intervals at small cost to coverage. For almost all lightly loaded hosts, the span increases slightly, driving the coverage slightly closer to the target level.

In essence, the MEAN predictor typically produces confidence intervals with larger than needed spans, resulting in more coverage than is requested. In contrast, the LAST predictor typically produces insufficiently large spans, resulting in smaller than requested coverage. The AR(16) predictor typically operates between these two extremes, producing appropriate coverage with small span. On some high load machines, the behaviors of MEAN and LAST are reversed, but AR(16) still produces the best results.

Effect of nominal time on confidence interval quality

The performance of the running time advisor depends on the nominal time of the task, t_{nom}. To illustrate this dependence, we use the same methodology to mine the testcases as described in the previous section, but we condition the results based on t_{nom}. Recall that the range of t_{nom} is from 0.1 to 10 seconds. In the previous section, to produce a point on a graph, we used testcases from this entire range. In this section, we divide the range of t_{nom} into three subranges: 0.1 to 3 seconds ("small tasks"), 3 to 6 seconds ("medium tasks"), and 6 to 10 seconds ("large tasks"). Then, for each of these subranges we produce a graph. Where previously we had one graph where each point represented the average of approximately 1000 testcases, we
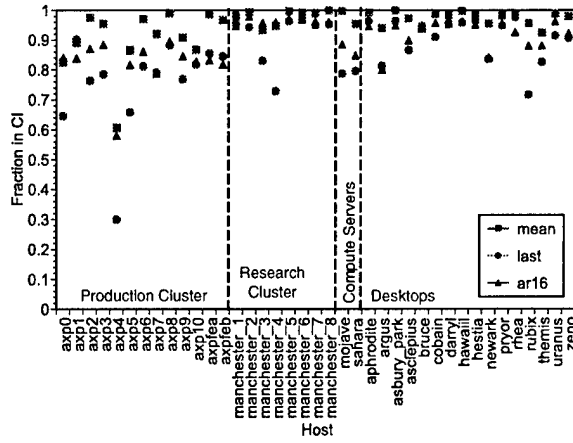
(a) Coverage, 0.1 to 3 second tasks
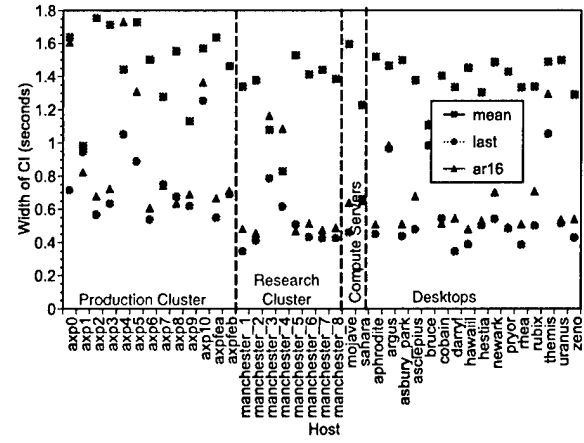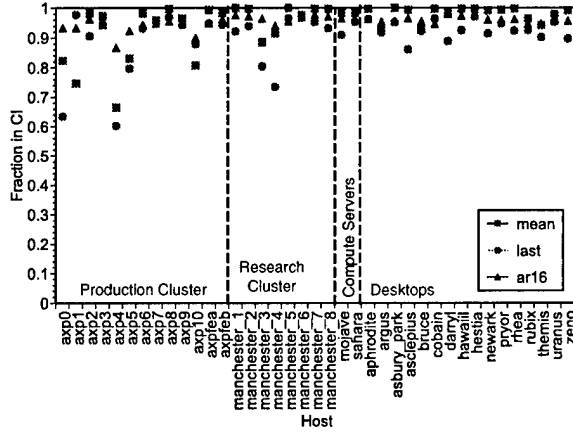
(b) Span, 0.1 to 3 second tasks

(c) Coverage, 3 to 6 second tasks

(d) Span, 3 to 6 second tasks

(e) Coverage, 6 to 10 second tasks

(f) Span, 6 to 10 second tasks

Figure 5.8: Effect of nominal time on confidence interval metrics, all hosts.

(a) Coverage, 0.1 to 3 second tasks

(b) Span, 0.1 to 3 second tasks

(c) Coverage, 3 to 6 second tasks

(d) Span, 3 to 6 second tasks

(e) Coverage, 6 to 10 second tasks

(f) Span, 6 to 10 second tasks

Figure 5.9: Effect of nominal time on confidence interval metrics, all hosts, LAST and AR(16) compared to MEAN.
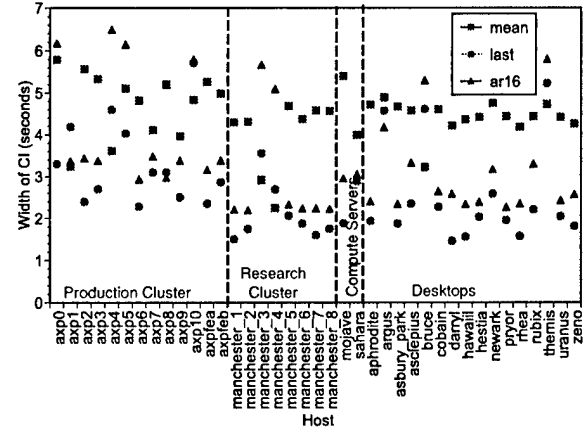
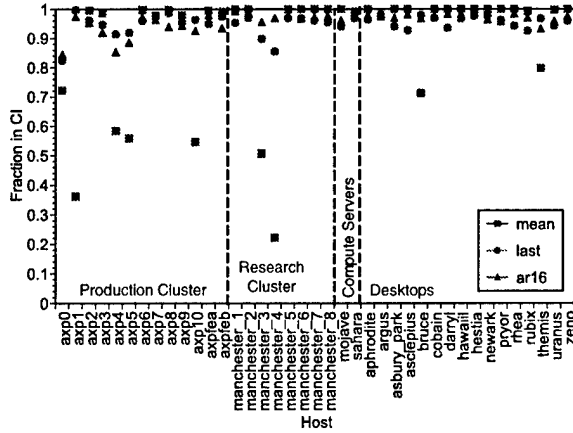(a) Coverage, 0.1 to 3 second tasks
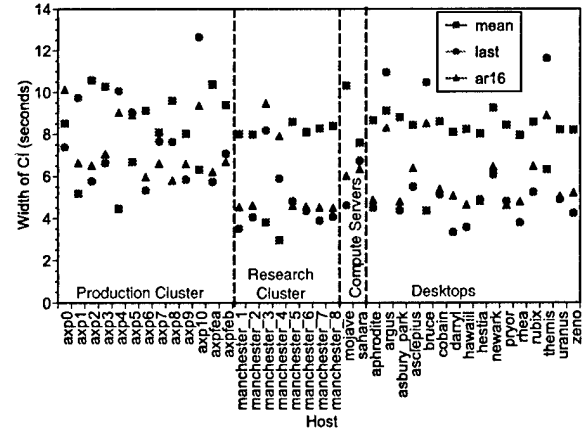
(b) Span, 0.1 to 3 second tasks

(c) Coverage, 3 to 6 second tasks
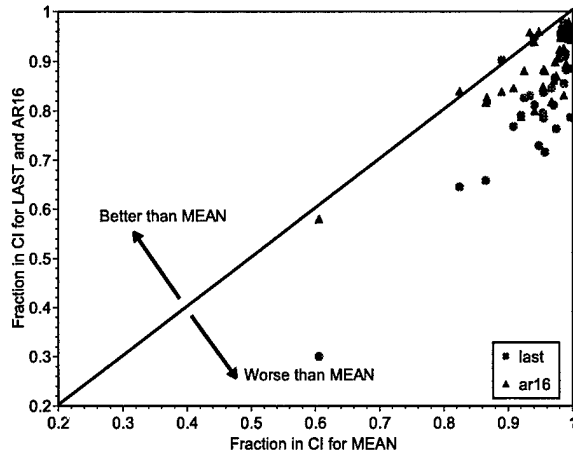
(d) Span, 3 to 6 second tasks

(e) Coverage, 6 to 10 second tasks

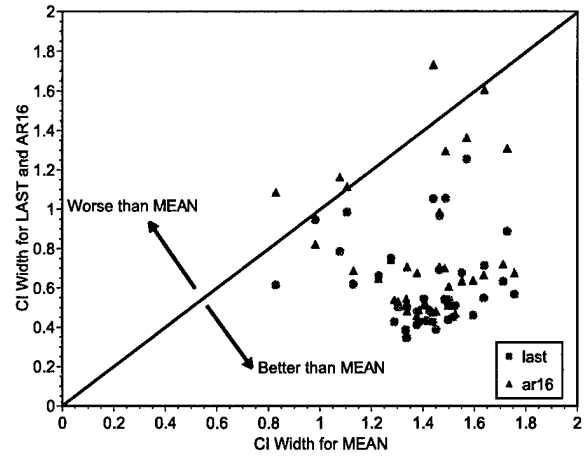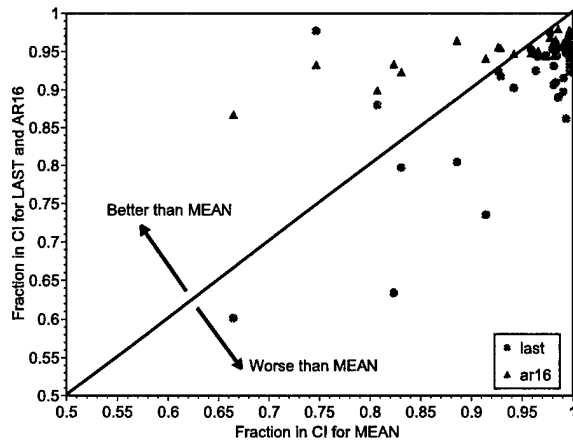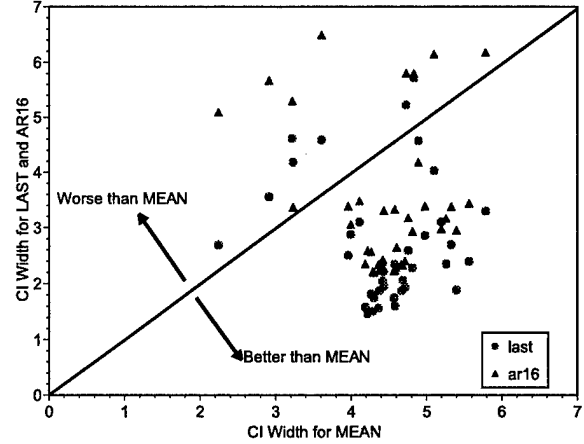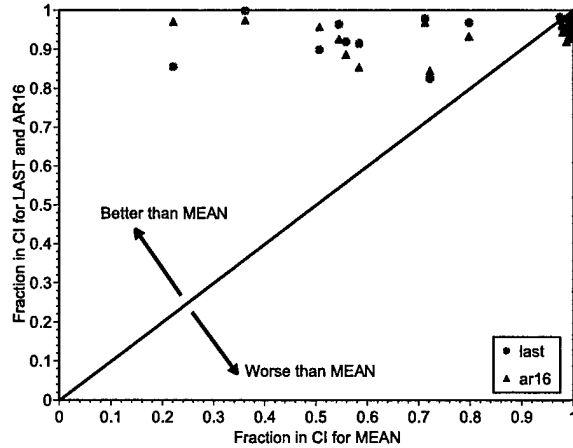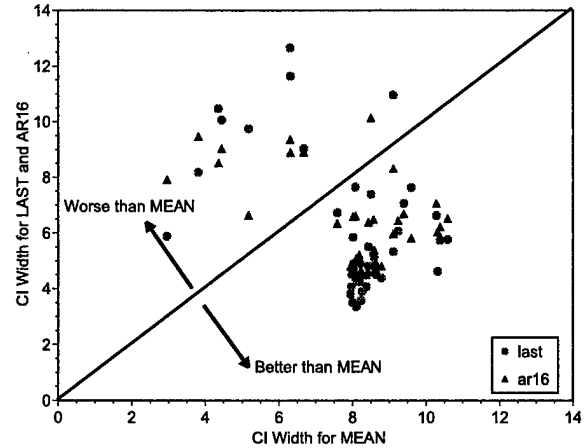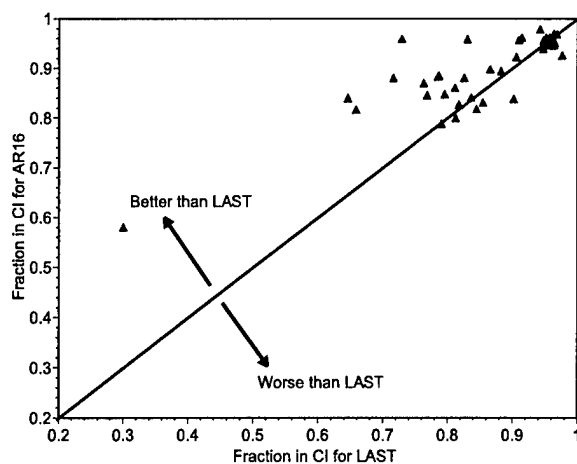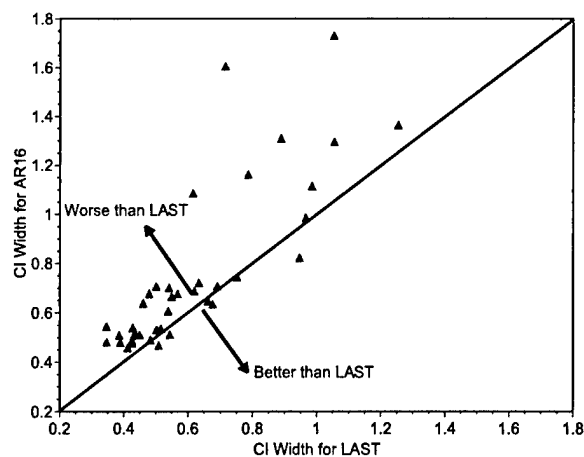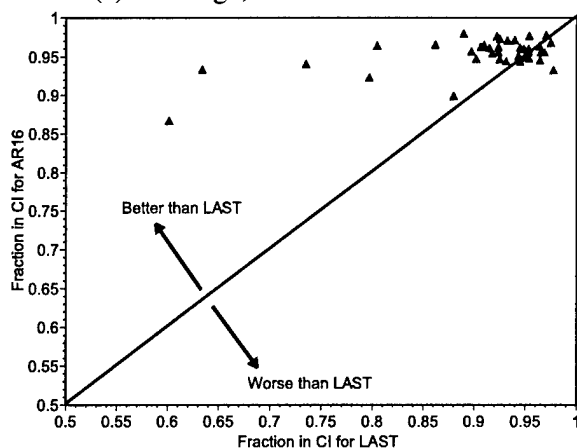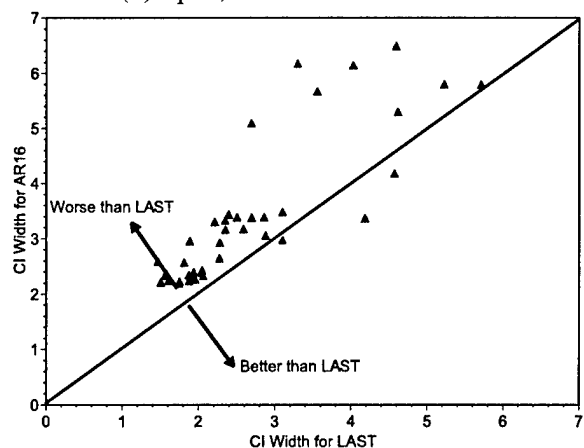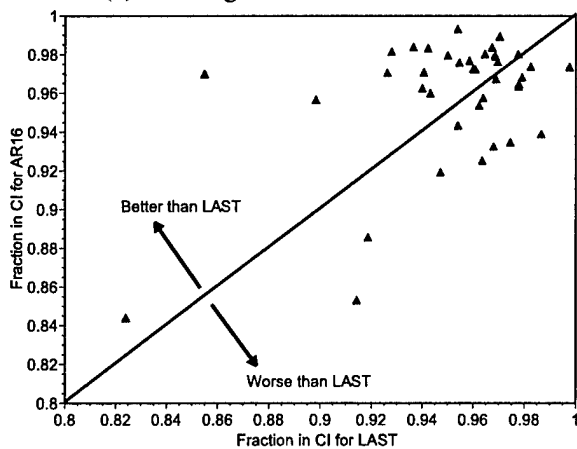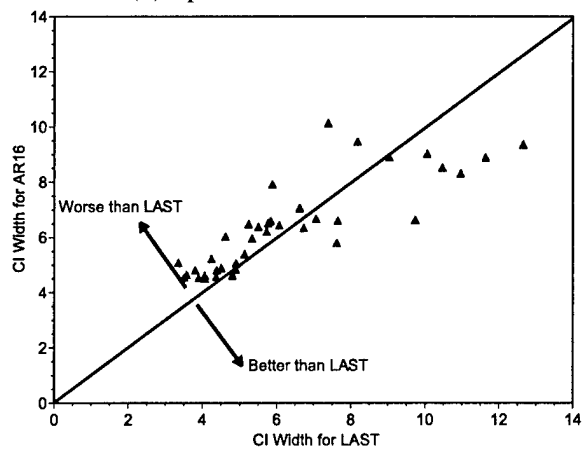(f) Span, 6 to 10 second tasks

Figure 5.10: Effect of nominal time on confidence interval metrics, all hosts, AR(16) compared to LAST.

now will have three graphs and a point on a graph represents the average of approximately 333 testcases in the appropriate range. The semantics of the graphs are as before. In the next section, we will delve deeper, dividing the hosts into five classes and illustrating how performance depends on nominal time for individual representative hosts.

Figures 5.8, 5.9, and 5.10 illustrate how the coverage and span of confidence intervals depend on the nominal time. The first figure presents the performance metrics of the three predictors on each of the traces for each of the three sizes of tasks. Figures 5.8(a) and (b) show the coverage and span for tasks of 0.1 to 3 second nominal times, (c) and (d) for 3 to 6 second tasks, and (e) and (f) for 6 to 10 second tasks.

In terms of the coverage, Figure 5.8 appears to show that all of the predictors do better as the nominal time increases. For short 0.1 to 3 second tasks (Figure 5.8(a)), there are a number of cases where none of the predictors provide better than 90% coverage, while for 6 to 10 second tasks (Figure 5.8(e)), there is almost always some predictor which provides a span of over 95%. In terms of the span, there appears to be a relative decrease in the gain of the more sophisticated predictors.

Figure 5.9 replots the data to compare the LAST and AR(16) predictors to the MEAN predictor while Figure 5.10 compares the AR(16) predictor with the LAST predictor. This illustrates the benefits of moving to the more sophisticated predictors. The benefit clearly increases as the nominal time increases. For example, at 0.1 to 3 seconds (Figure 5.9(a) and (b)) we can see that the more sophisticated predictors actually have worse coverage than the simple MEAN predictor because their spans are too narrow. In general, however, we can see that the AR(16) predictor does better than LAST (Figure 5.10(a) and (b)), producing wider spans that result in better coverage. However, this is clearly a difficult case, since even with AR(16), the coverage on half of the traces is less than 80%.

As the nominal time increases (Figure 5.9(c)–(f), Figure 5.10(c)–(f)) we begin to see a picture that is more similar to the overall picture of Section 5.4.3. On the heavily loaded hosts, the more sophisticated predictors produce wider spans which result in much better (and more appropriate) coverage. On the lightly loaded hosts, the coverage is similar between the different predictors, but LAST and AR(16) produce much narrower spans. In comparing LAST and AR(16), we can see that as the nominal time increases the relationship between these two predictors becomes more complex. For medium sized tasks (Figure 5.10(c)-(d)), AR(16) produces large gains in coverage over LAST, bringing coverage to the target 95% levels in many cases. The cost is a typically small increase in span, although the difference is larger for more heavily loaded hosts. For large tasks (Figure 5.10(e)-(f)), LAST and AR(16) seem to be in a dead heat.

Nominal time independent evaluation of quality of expected running times

This section evaluates how well the expected running time, t_{exp}, provided by our algorithm predicts the actual running time, t_{act}, encountered when running the task. We will begin by using the same methodology as the previous sections, but with the R^2 metric. We will also discuss certain important characteristics that are not measured by R^2

Figure 5.11 shows the R^2 values measured on each of the traces using each of the predictors. Each point represents approximately 1000 testcases. As we can see, in almost all cases, some predictor provides an $R^2 > 0.9$—Our task running time predictions explain over 90% of the variability in the running time. There seems to be little distinction between the different predictors, however.

Figure 5.12 plots the R^2 values of the LAST and AR(16) predictors versus their corresponding values for the MEAN predictor. Roughly half of the hosts benefit from the more sophisticated predictors. Among these are the more heavily loaded hosts. The other half of the hosts do not benefit from the more sophisticated predictors, but their loss is not as significant as the gains of the half that do. It is also clear that the AR(16) predictors generally have very little loss compared to MEAN.

Figure 5.13 compares the R^2 values of the LAST and AR(16) predictors. About 2/3 of the hosts see
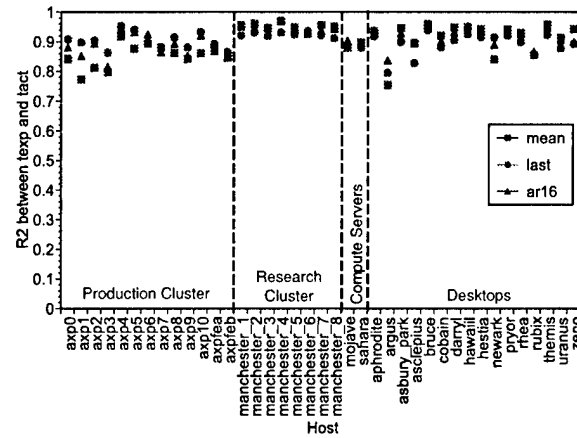
Figure 5.11: R^2 metric for 0.1 to 10 second tasks on all hosts, independent of nominal time.
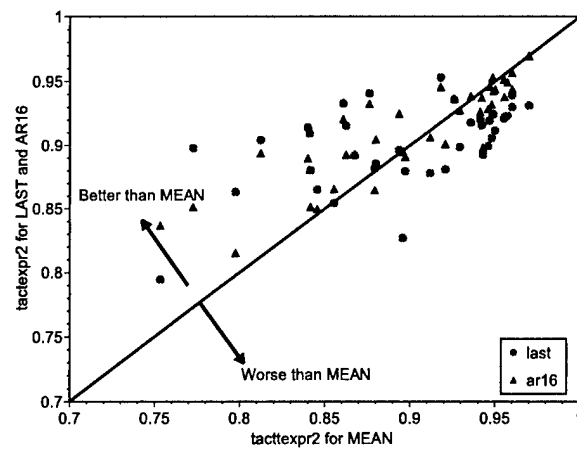
Figure 5.12: R^2 metric showing LAST and AR(16) versus MEAN for 0.1 to 10 second tasks on all hosts, independent of nominal time.
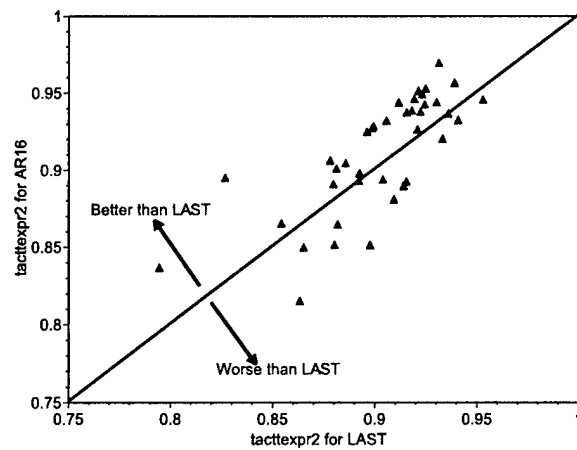
Figure 5.13: R^2 metric showing AR(16) versus LAST for 0.1 to 10 second tasks on all hosts, independent of nominal time.
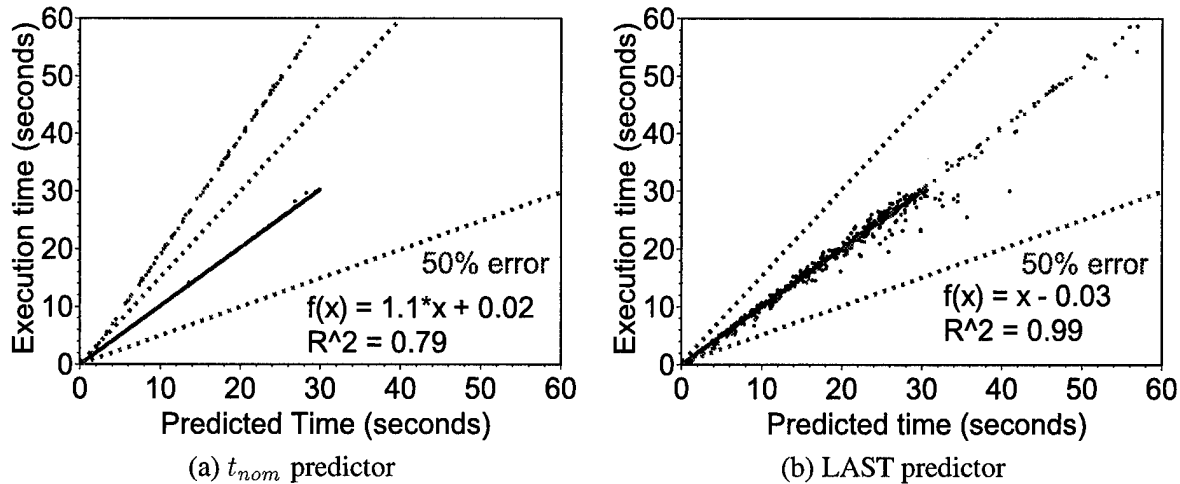
Figure 5.14: Actual versus expected running times for lightly loaded host with intermittent load.

some benefit from the move to the AR(16) predictor, while the rest see some detriment. The gain or decline is on the order of 0.05 or so.

It turns out that most of the variation in the running time of tasks can be explained by the nominal time of the task itself. Typically, the R^2 using the nominal time of as predictor is on the order of 0.8, meaning that 80% of the variation in the running time can be explained simply by the nominal time of the task, t_{nom}. Because of this there is little room for improvement by the predictor, which is why the predictors have similar performance.

This is not to say that the nominal time is a good overall predictor, because it can be very sensitive to transient behavior that a predictor based on host load can more appropriately deal with. Furthermore, the R^2 value shows how much of a linear relationship there is between the actual running time and the prediction, but it does not give any insight into what the slope and intercept of this relationship are. Ideally, a good predictor would have $t_{act} = mt_{exp} + b + noise$ where $m = 1$ and $b = 0$. Intuitively, the value of m is determined by how well the predictor captures the contribution of the load to the running time.

Figure 5.14 illustrates how a more sophisticated predictor can avoid transient behavior, resulting in significantly higher R^2. Each of the graphs plots 1000 tasks chosen from 0.1 to 30 seconds, plotting their actual running time versus the predicted running time, determined either as (a) the nominal time t_{nom} or (b) by using the LAST predictor. We have also plotted a 50% error region around the predictions. Because the host had a background process that ran at infrequent times, the t_{nom} predictor turned out to be wildly inaccurate for about 10% of the tasks, under-predicting running times by a factor of 2. The LAST predictor, which incorporates the most recent host load measurements, was able to detect when the background process ran, offering higher predicted running times when this was the case. Because of this, the R^2 rose from 0.79 to 0.99. More importantly, the linear fit has improved from $m = 1.1$ and $b = 0.02$ to $m = 1$ and $b = -0.03$.

Figure 5.15 shows another typical behavior. In this case we have run approximately 3000 tasks ranging from 0.1 to 10 seconds and have plotted them as per the previous figure. The host was playing back the high load axp0 trace. In Figure 5.15(a) we see that over 74% of the t_{nom} predictions are wrong by more than 50%. On the other hand, using the AR(16) predictor (Figure 5.15(b)) only 3% of the predictions are wrong by more than 50%. However, note that the R^2 value has changed only marginally, from 0.82 to 0.87—the linear fit to the t_{nom} graph is not much worse than that of the AR(16) predictor. However, the structure of the fit has improved from $m = 2.15$ and $b = 0.82$ to $m = 0.95$ and $b = 0.06$.

(a) t_{nom} predictor   (b) LAST predictor
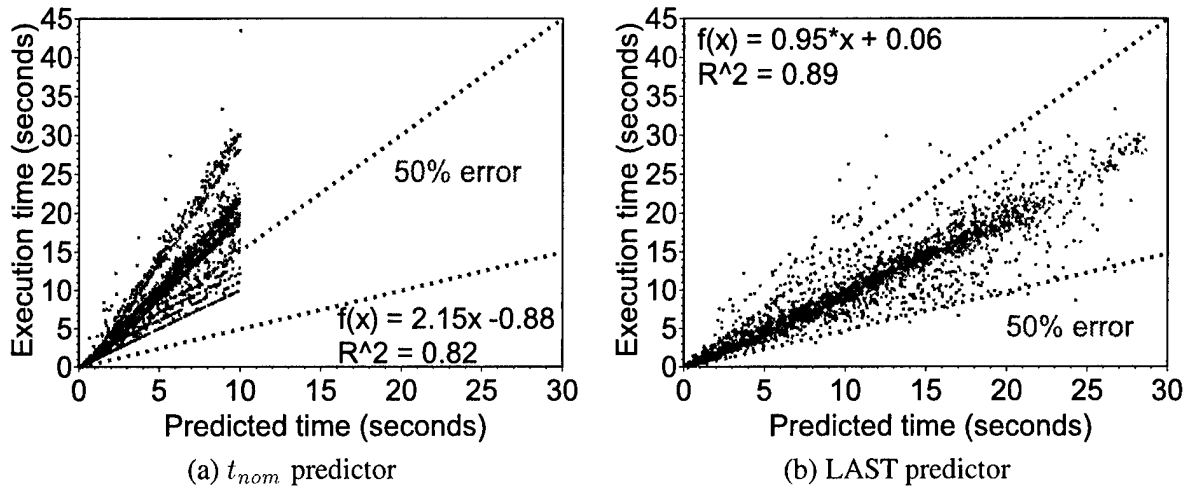
Figure 5.15: Actual versus expected running times for heavily loaded host (axp0 trace).

Effect of nominal time on quality of expected running times

The R^2 of a prediction is strongly dependent on the nominal time of the task, but is somewhat independent of which host load predictor is used. Figure 5.16 illustrates how the average R^2 value measured for each of the load traces depends on the nominal time. As can be seen from the figure, R^2 values decline significantly as tasks grow in size. For small tasks (Figure 5.16(a)), there is generally some predictor which provides $R^2 > 0.9$, while for large tasks (Figure 5.16(c)), most R^2 values are significantly below 0.8. Furthermore, as tasks increase in size, there seems to be greater variability in the performance of the different predictors.

This variability in performance among the predictors is not, alas, evidence that one of the predictors is consistently better than the others for all, or even a large group of, hosts. Indeed, Figure 5.17, which compares the R^2 of the LAST and AR(16) predictors to that of the MEAN predictor, and Figure 5.18, which compares the R^2 of the LAST and AR(16) predictors, show that there are no clear favorites. As running times increase, R^2 values decline precipitously and no clear pattern of difference among the hosts becomes clear—no one predictor is preferable. If R^2 is the desired figure of merit, then a multiple expert approach, where different predictors are used simultaneously, and the predictor with the best recent predictions is the one whose value is reported to the user.

Host classes

For each individual load trace, we plotted our three performance metrics (coverage, span, and R^2) versus the nominal time t_{nom}. When we did this, we found that an interesting pattern emerged. By visual inspection, the results for the 39 traces could be placed into five classes. At present, we make no use of these categorizations, and we certainly have not automated the characterization process. Nonetheless, examining representatives of each of the classes is enlightening and permits us to make recommendations for each of the classes.

Class I : Class I, which we also call the "typical low load host" class represents the most common behavior by far that we have encountered. The class consists of 29 of the 39 hosts (76%): axp2, axp3, axp6, axp7, axp8, axp9, axpfea, axpfeb, manchester-1, manchester-2, manchester-5, manchester-6, manchester-7, manchester-8, mojave, sahara, aphrodite, asbury-park, asclepius, cobain, darryl, hestia, hawaii, newark, pryor, rhea, rubix, uranus, and zeno.

Class I is exemplified by trace axp2, whose metrics we plot as functions of the nominal time in Fig-
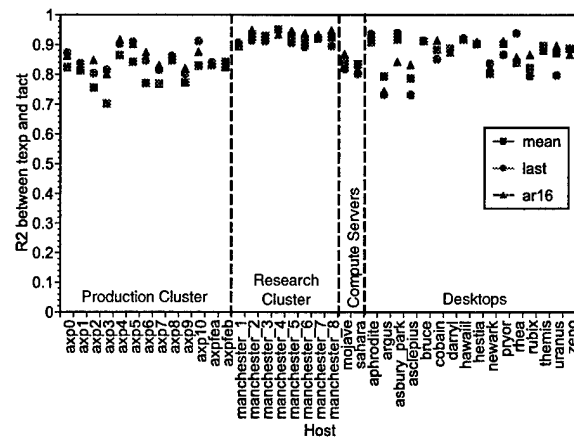
(a) R^2, 0.1 to 3 second tasks
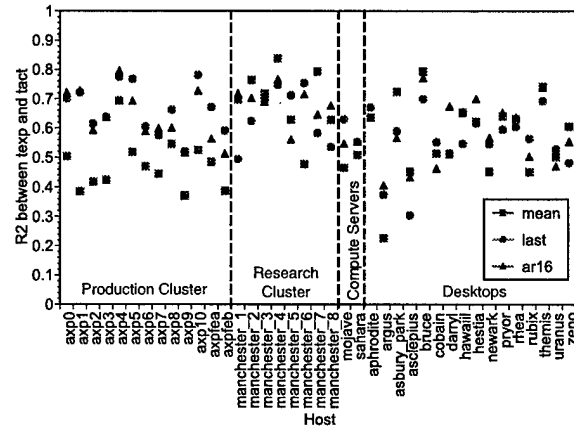
(b) R^2, 3 to 6 second tasks
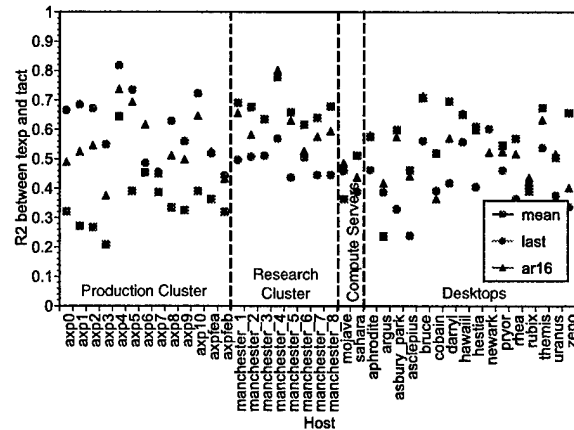
(c) R^2, 6 to 10 second tasks

Figure 5.16: Effect of nominal time on R^2 metric, all hosts.
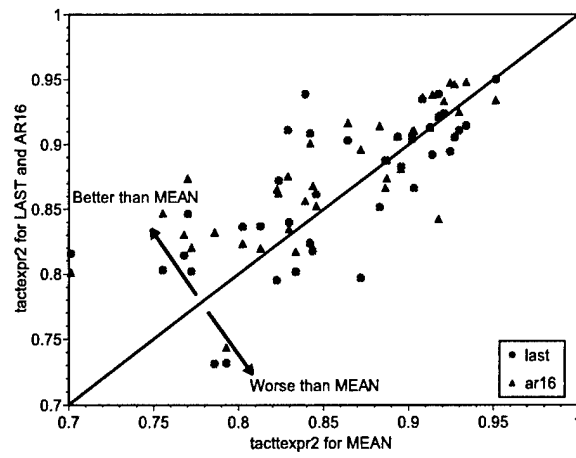
(a) R^2, 0.1 to 3 second tasks
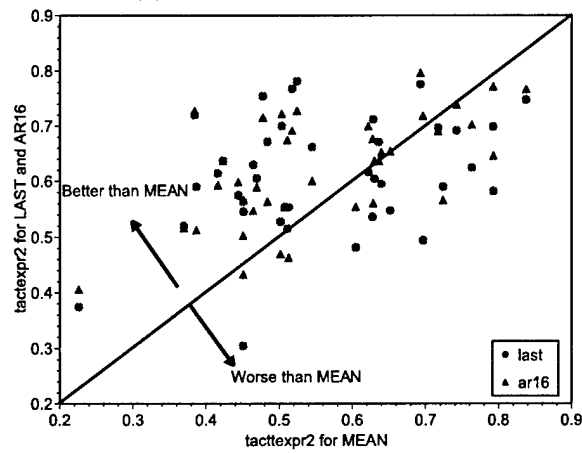
(b) R^2, 3 to 6 second tasks
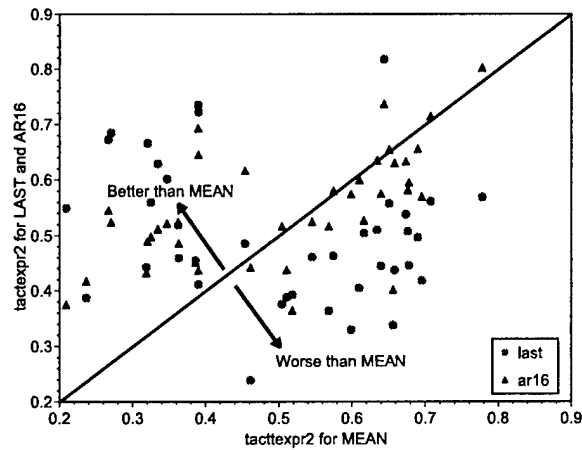
(c) R^2, 6 to 10 second tasks

Figure 5.17: Effect of nominal time on R^2 metric, all hosts, LAST and AR(16) compared to MEAN.
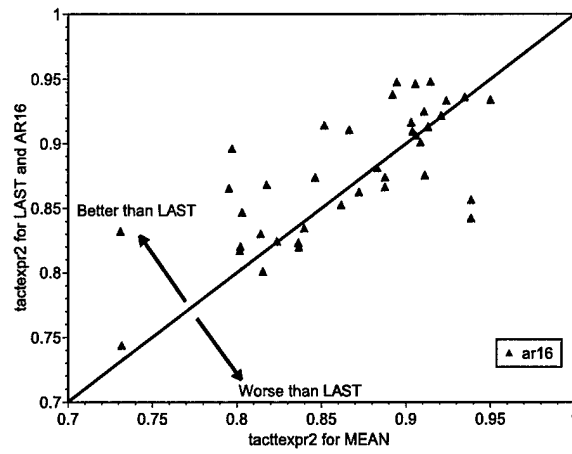
(a) R^2, 0.1 to 3 second tasks
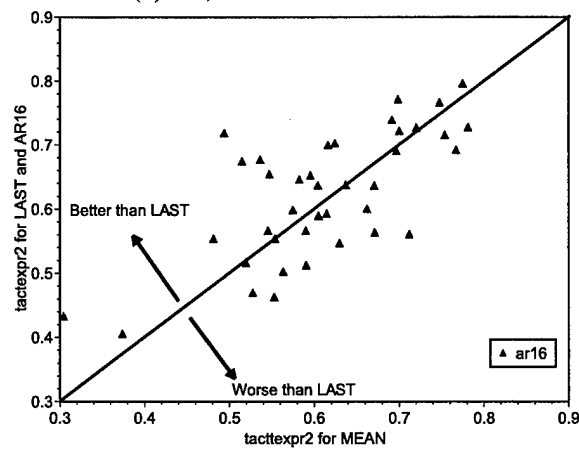
(b) R^2, 3 to 6 second tasks
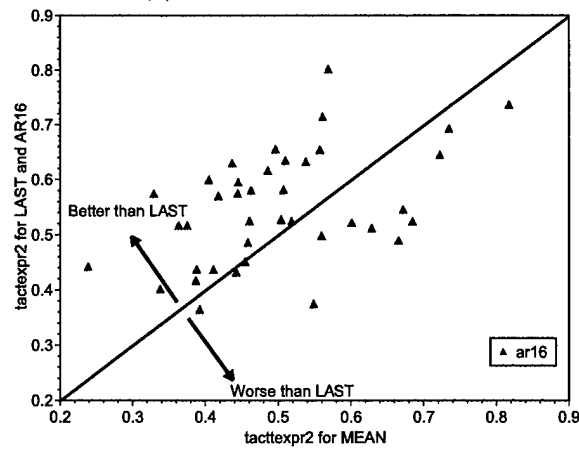
(c) R^2, 6 to 10 second tasks

Figure 5.18: Effect of nominal time on R^2 metric, all hosts, AR(16) compared to LAST.
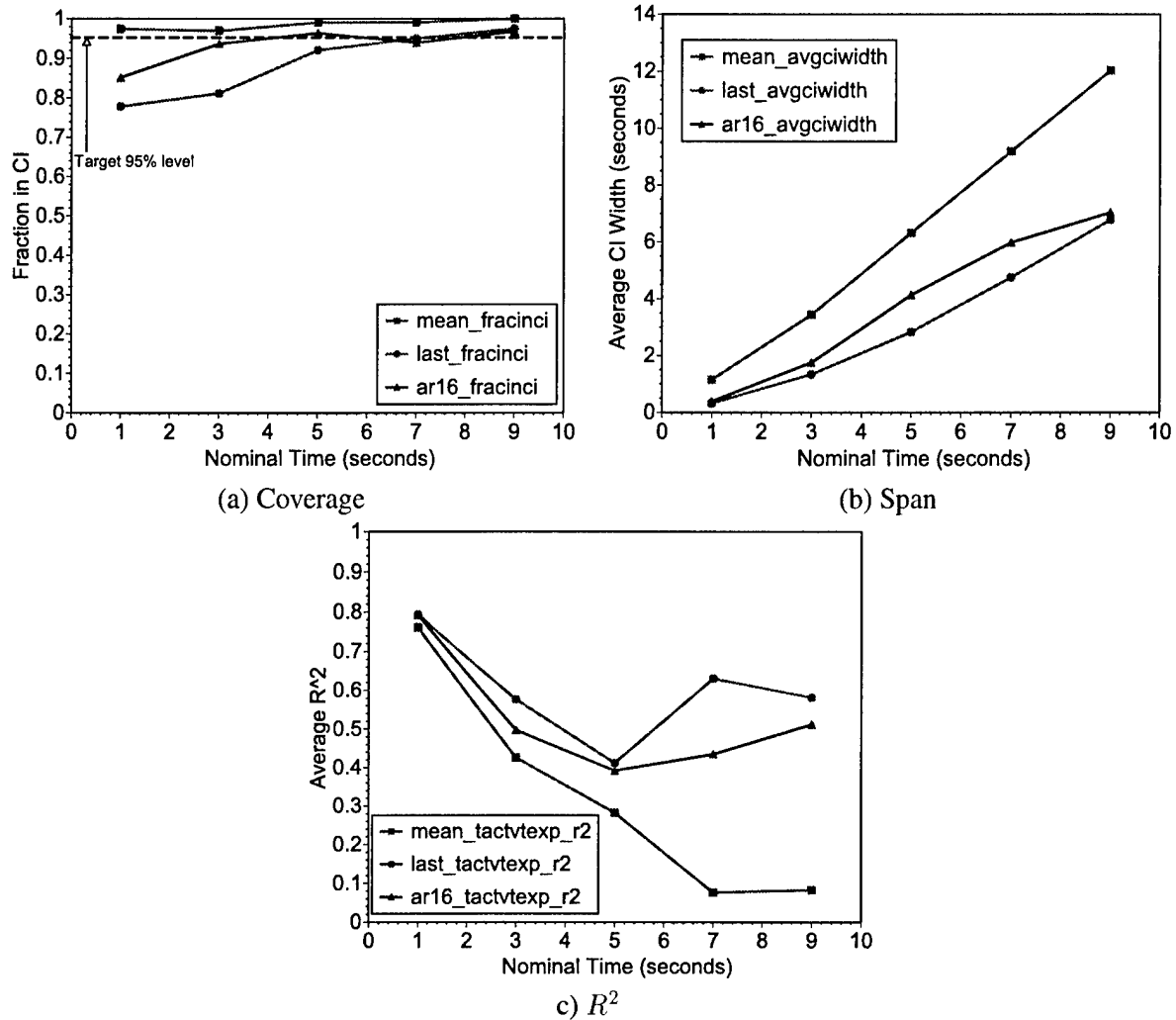
Figure 5.19: Performance metrics as function of time for Class I ("typical low load host") host axp2.

ure 5.19. Each point in the graph represents the average of about 200 testcases and represents a 2 second span of nominal time, extending from one second before the point's x-coordinate to one second after. The main characteristics of the class are the following. The coverage is only slightly dependent on the nominal time, increasing slightly for all predictors as the nominal time increases. The MEAN predictor typically has almost 100% coverage and is closely followed by the AR(16) and then the LAST predictor. The LAST and AR(16) predictors have significantly narrower spans than the MEAN predictor, with AR(16) producing slightly wider spans than LAST. The R^2 drops precipitously with increasing nominal time, especially for MEAN. The drop is less precipitous for LAST and AR(16), and LAST is usually slightly better than AR(16) in terms of R^2. For this particular trace, there is a slight upturn for LAST and AR(16) with increasing R^2, while in in other cases the decline is monotonic.

Our primary concern is producing accurate confidence intervals. For this reason, we believe that the AR(16) is the best predictor for this class of host. The coverage is nearly as good as MEAN and is typically near the target 95% point, while LAST tends to lag behind, especially for smaller tasks. Furthermore, the span of AR(16) is typically half that of MEAN and only slightly wider than LAST. In most hosts, then, a better predictor produces much narrower accurate confidence intervals.

Class II : Class II hosts, which we refer to as being in the "atypical low load host" class, present the

(a) Coverage   (b) Span

c) R^2
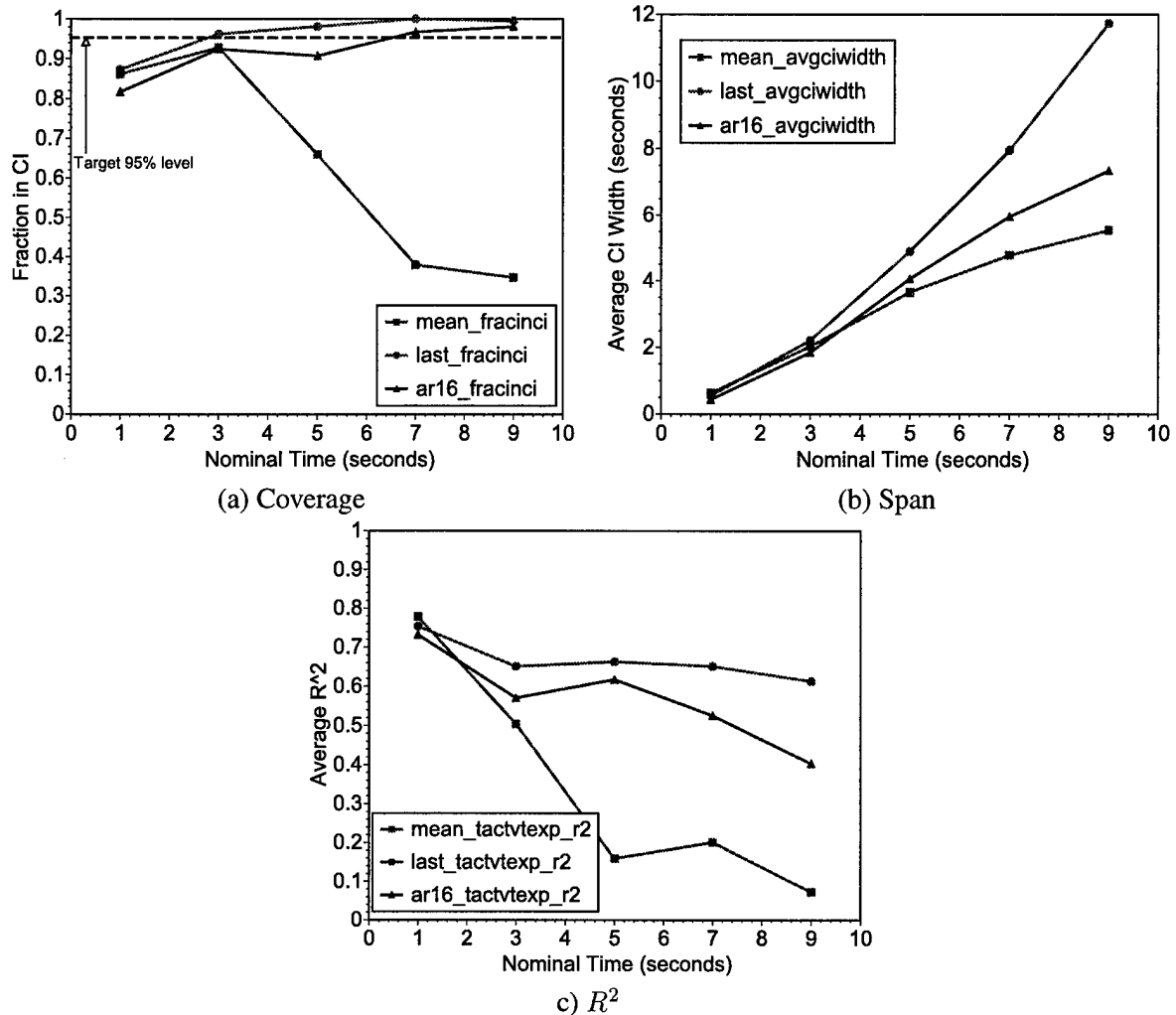
Figure 5.20: Performance metrics as function of time for Class II ("atypical low load host") host axp1.

second most common behavior among our traces. The class consists of 4 of the 39 hosts (10%): axp1, manchester-3, manchester-4, and bruce.

An exemplar of Class II is the trace axp1. We have plotted the metrics of axp1 in Figure 5.20. The methodology behind the graphs is identical to that of Class I. An important distinguishing feature of this class is that the coverage of the MEAN predictor drops precipitously with increasing nominal time because the span of its confidence interval is not sufficiently large. In contrast, LAST and AR(16) compute slightly larger confidence intervals which result in excellent coverage that increases with increasing nominal time. LAST and AR(16) have similar coverage (in this case LAST is slightly ahead, in other cases AR(16) is slightly ahead). The R^2 of MEAN also decays quickly with increasing nominal time, while those of LAST and AR(16) decay much more slowly. Again, in some cases, AR(16) is slightly ahead, in others LAST is slightly ahead by this metric.

In terms of computing confidence intervals, either AR(16) or LAST seems adequate for producing confidence intervals for this class of host. Compared to MEAN, both produce significantly larger spans that result in much better coverage. In terms of the expected time, AR(16) and LAST are also to be strongly preferred over the MEAN predictor, while there is no clear advantage to recommending one over the other.

Class III : The remainder of the five host classes all contain high load hosts. There does not seem to

(a) Coverage   (b) Span

(c) R^2
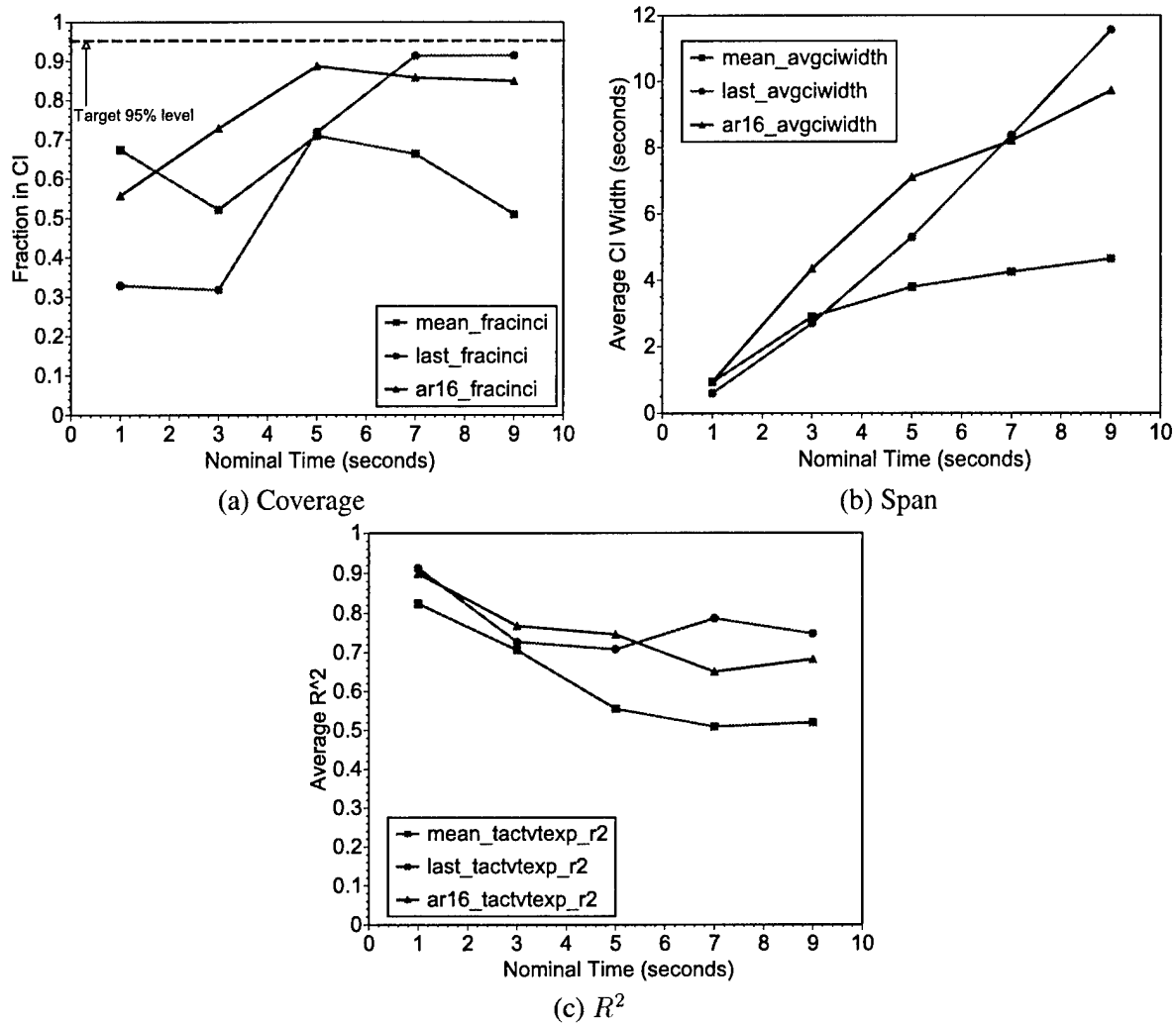
Figure 5.21: Performance metrics as function of time for typical class III ("high load 1") host axp4.

be a "typical" behavior on a high load host, so we will simply enumerate these classes. Class III, which we also call "high load 1", consists of a 3 of the 39 hosts (8%) : axp4, axp5, and argus.

Using the same methodology as before, Figure 5.21 plots the performance metrics as a function of the nominal time for an exemplar, axp4. Compared to the low load hosts, this high load 1 host displays much more complex behavior. The predictor with the best coverage depends strongly on the nominal time. For very short tasks, MEAN is slightly better than AR(16), which is much better than LAST, although the coverage is quite poor with all three predictors. For medium size tasks, AR(16) provides the best coverage, followed at a distance by MEAN and LAST, which become interchangeable. For large tasks, AR(16) and LAST have similar coverage, with AR(16) lagging slightly, while MEAN's coverage is far behind. In terms of the span, AR(16) and LAST both compute much wider confidence intervals than MEAN, which explains why their coverage is so much better. MEAN is unable to understand the dynamicity of this kind of host. Predictably, for the nominal times where AR(16) is preferable to LAST, it has a larger span. In terms of the R^2, AR(16) and LAST are clearly preferable to MEAN since their performance declines much more gradually with increasing nominal time. They are interchangeable.

In terms of computing accurate expected running times for this class of hosts, either AR(16) or LAST are appropriate since their similar R^2 metrics are significantly better than that of MEAN. In terms of computing
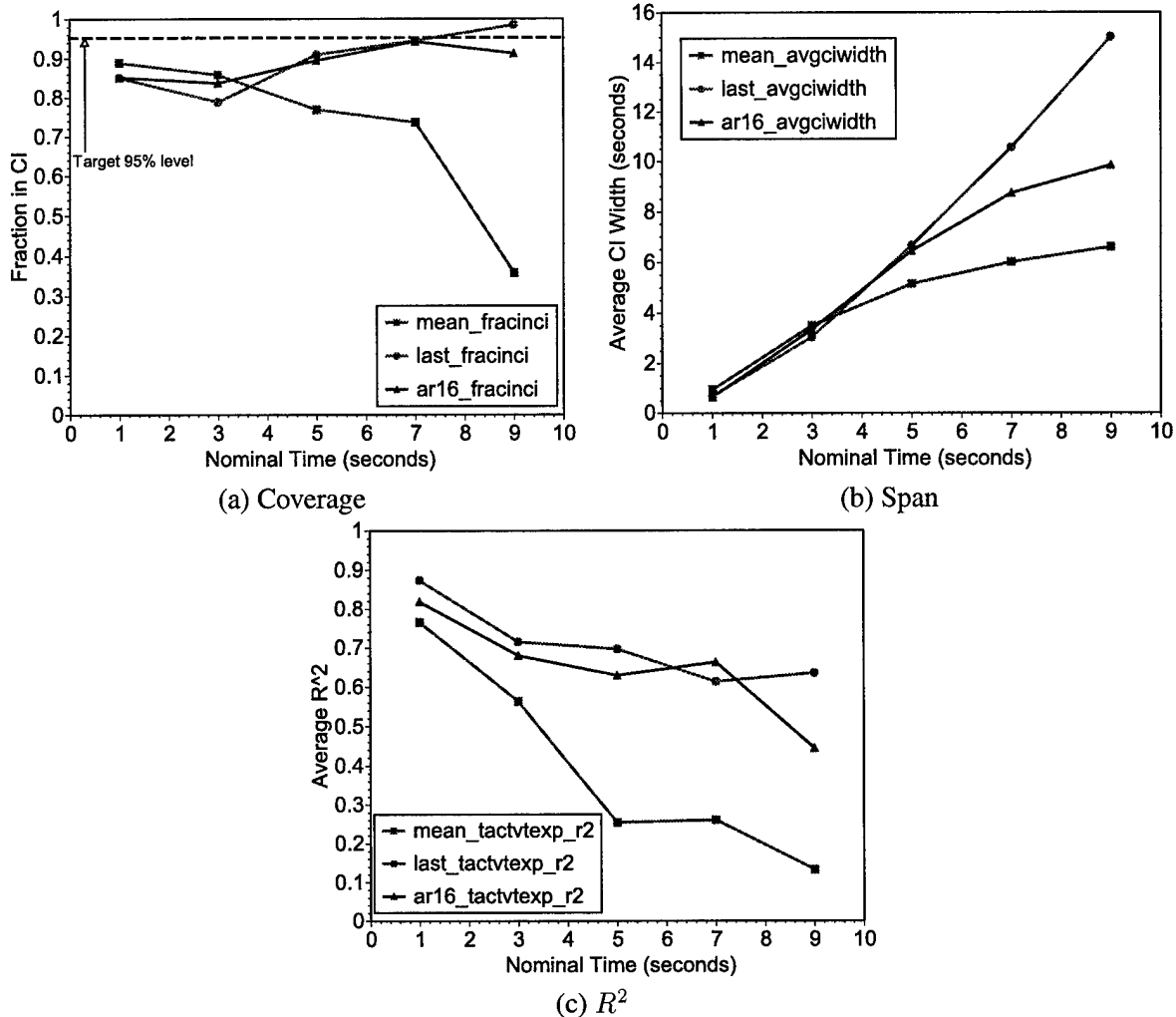
Figure 5.22: Performance metrics as function of time for typical class IV ("high load 2") host axp10.

accurate confidence intervals, the best predictor is highly dependent on the nominal time. For very short tasks, MEAN or AR(16) is preferable, but either has rather poor coverage. For medium tasks, AR(16) produces the best performance. For large tasks, LAST is best.

Class IV : This class, which we also refer to as the "high load 2" class, contains two hosts: axp10 and themis.

Figure 5.22 plots the performance of the predictors on a representative trace, axp10, using the same methodology as before. We can see that the coverage of LAST and AR(16) are virtually identical here and increase slowly with nominal time. MEAN has similar coverage for small tasks, but then behaves increasingly poorly, with coverage decreasing rapidly with nominal time. In terms of the span, LAST grows much more quickly than MEAN with increasing nominal time, while AR(16) is almost exactly in between them. For very short nominal times the spans are all identical. The R^2 of MEAN drops in step with increasing nominal time, while the R^2 of LAST and AR(16), which remain virtually identical, drop much more slowly.

In terms of computing confidence intervals, AR(16) clearly produces the best results for this class of hosts, getting coverage identical to that of LAST with a span that is often half as wide. In terms of computing expected times, AR(16) and LAST are nearly identical, and significantly better than MEAN.

(a) Coverage

(b) Span

(c) R^2
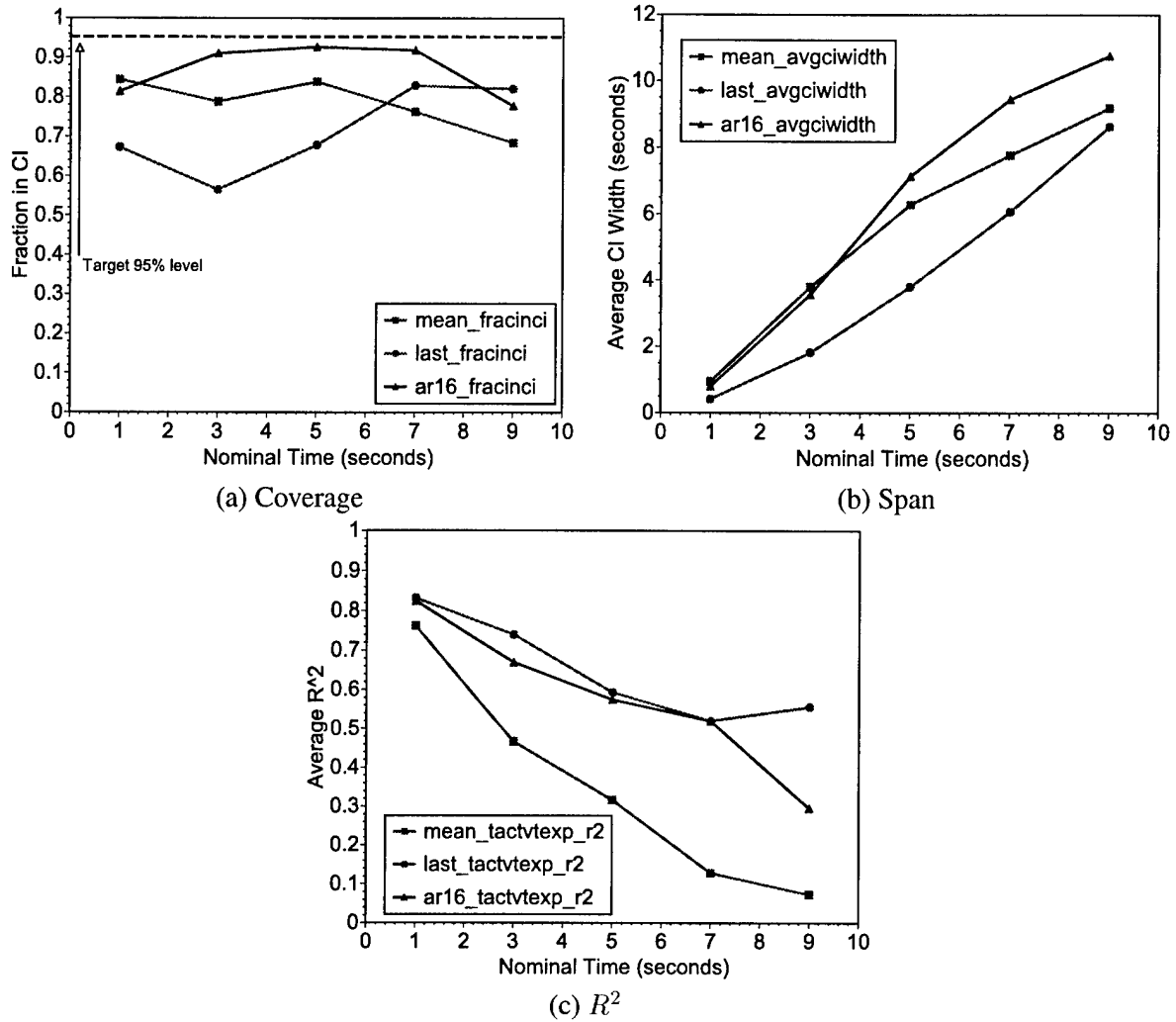
Figure 5.23: Performance metrics as function of time for typical class V ("high load 3") host axp0.

Class V : Class V, which we also refer to as the "high load 3" class, consists of a single host, axp0.

Figure 5.23 plots the performance of the predictors on axp0 using the same methodology as before. In terms of coverage, AR(16) is clearly the winner here, especially for medium sized tasks. It achieves its reasonable coverage (the goal is 95%) by computing slightly larger confidence intervals than MEAN. LAST computes confidence intervals that are far too small, resulting in abysmal coverage. In terms of R^2, LAST and AR(16) provide nearly identical performance that drops much more gradually with increasing nominal time than that of MEAN.

AR(16) is clearly the preferable predictor for this class of hosts in terms of computing confidence intervals. For the expected time, AR(16) and LAST are interchangeable here.

5.5 Conclusion

We began this chapter with a resource prediction service capable of providing accurate host load predictions. However, applications and application schedulers are interested in application-level predictions. In particular, the real-time scheduling advisor needs predictions of task running time. To provide these predictions,

we developed an algorithm for transforming from host load predictions and a task's nominal time, a measure of the CPU demand of the task, into a prediction of the task's running time. The prediction is expressed both as a point estimate (the expected running time) and a confidence interval for the running time.

We implemented the algorithm to provide a new service, the running time advisor, on top of the existing host load prediction service. We constructed a sophisticated run-time evaluation infrastructure based on the new technique of load trace playback and evaluated our implementation using it. The main result is that the system works well when combined with an appropriate host load predictor, such as the AR(16) predictor that we found most appropriate when we evaluated host load prediction in the previous chapter.

Chapter 6

Real-time Scheduling Advisor

This dissertation argues for basing real-time scheduling advisors on explicit resource-oriented prediction, specifically on the prediction of resource signals. In the previous chapters, we presented a methodology and tools for understanding such signals and developing prediction systems for them. We then applied the approach to host load and developed an appropriate host load prediction system. In the last chapter, we demonstrated how to estimate the running time of a task using these host load predictions, and implemented and evaluated a tool, the running time advisor, that does so. The running time advisor provides information that can be used in many forms of application-level scheduling. This chapter addresses one such form: real-time scheduling for interactive applications.

The real-time scheduling advisor is straightforwardly implemented on top of the running time advisor. We implemented prediction-based strategies based on the MEAN, LAST and AR(16) predictors of the last chapter. The prediction-based strategies choose randomly from among the hosts whose running time predictions are appropriate for meeting the deadline with the confidence that the application desires. In addition to advising the application to run the task on a particular host, the prediction-based strategies also specify whether they believe the deadline can actually be met on that host. This allows the application to negotiate with the advisor to find a combination of CPU demand and deadline such that the deadline can confidently be met.

We also considered, for comparison purposes, two strategies that are not based on prediction. The first, RANDOM, randomly assigns tasks to hosts, and is clearly of lowest overhead. The second, MEASURE, assigns tasks to the host whose current load measurement is minimum. Compared to MEASURE, the prediction-based strategies have minimal additional overhead. Like the prediction-based strategies, RANDOM and MEASURE recommend a host to the application. However, unlike the prediction-based strategies, they can not determine whether or not the task is likely to meet its deadline on that host. Additionally, the MEASURE strategy is unable to introduce any randomness into its scheduling decisions, which increases the chance of disastrous synchronization among MEASURE-based advisors.

While the real-time scheduling advisor is simple, its evaluation is complex. We evaluated the performance of the five scheduling strategies using three different metrics that measure how likely a deadline is to be met, how trustworthy the advice of an advisor is, and how much randomness an advisor can introduce into its decisions. These metrics, especially the first, depend not only on the quality of predictions, but also on the characteristics of the set of available hosts, and how much slack the deadline allows. In particular, a missed deadline can be due to either a lack of resources (which is not the fault of the real-time advisor) or to prediction error (which is). To aid in understanding the results of our empirical evaluation, we developed an analytic model that relates the characteristics of the available hosts and the slack to the scheduling feasibility (the probability that there is a host on which the deadline can be met) and the predictor sensitivity (the probability that a deadline is missed due to a bad prediction). While the model is not predictive, it is useful

to help explain the sometimes counter-intuitive empirical results.

The evaluation of the scheduling strategies is based on running randomized testcases using a simple extension to the experimental infrastructure described in the previous chapter. In the evaluation, the hosts to which tasks can be assigned each play back one of the August, 1997 load traces. We evaluated the scheduling strategies using several different sets of traces, or *scenarios*, which were derived in part by clustering the traces by their statistics (ie, trying to explore the "basis" of the space of observed traces) and in part by approximating scenarios that might be seen in practice. Using the randomized testcases, we then can then measure the performance metrics of the different strategies and see how they vary with the scenario, the CPU demand of the task, and the slack.

The results of the evaluation show the clear benefits of using a prediction-based strategy, and the AR(16)-based strategy in particular. These strategies are the most trustworthy from the point of view of the application in that when they assert that a deadline can be met by using the recommended host, there is a high probability that it will actually be met. For the AR(16) strategy, this probability is maximal, reasonably independent of the nominal time, slack, and scenario, and is usually close to the confidence level that the application requested. In other words, when it is possible to find a host that can meet the deadline, AR(16) is best able to do so.

The probability that a deadline will be met, irrespective of whether the strategy thinks it can be met, depends strongly on scenario, nominal time, and slack. An important observation is that near the "critical slack" of a scenario, the prediction-based strategies, particularly AR(16), significantly increase the chance of meeting the deadline. The critical slack corresponds to where the deadline is just large enough so that a task's deadline can be met in expectation. At slack values that are low (tight deadline) or high (loose deadline) compared to the critical slack, performance is insensitive to prediction and the better prediction-based strategies (AR(16)) do only as well as the MEASURE approach. The MEASURE approach significantly outperforms the RANDOM approach in almost all cases.

The prediction-based strategies introduce a degree of randomness into their scheduling decisions that is between the extremes of the MEASURE strategy (no randomness) and RANDOM (pure randomness). Essentially, they are able to transform slack (and low nominal times) into randomness in their host selections while also increasing the chances of meeting the deadline and increasing trustworthiness from the application's point of view. This randomness is important because it makes it difficult for different advisors, which are oblivious of each other, to become synchronized and thus simultaneously provide bad advice to their applications. The AR(16) strategy is generally able to introduce the most randomness. In a separate experiment, we studied multiple advisors scheduling to the same group of hosts and found no evidence of contention problems.

In summary, the prediction-based strategies, particularly AR(16), are able to increase the probability that a deadline will be met over the pure MEASURE strategy, especially near the critical slack. The MEASURE approach is in turn clearly preferable, in almost all cases, to the RANDOM strategy. Furthermore, the prediction-based strategies are able to quite accurately inform the application whether the deadline can be met, which makes them more trustworthy than either MEASURE or RANDOM, and enables the application to find a fulfillable combination of CPU demand and deadline. Also, the prediction-based approaches are able to introduce randomness into their scheduling decisions to avoid unintended and disastrous synchronization with other scheduling advisors. Given that all of these benefits are purchased at only a marginal overhead over the MEASURE strategy, the case for using a prediction-based strategy in real-time scheduling advisors is clear. Finally, because the AR(16) strategy provides the best overall performance of all of the these strategies, we recommend basing real-time scheduling advisors on this strategy.

6.1 Real-time scheduling advisor interface

The interface that the real-time scheduling advisor provides to applications and application schedulers is very simple. The application provides the CPU demand of the task it seeks to schedule at the current time, the deadline of the task, the confidence with which it wants that deadline to be met, and a list of prospective hosts on which the task could be scheduled. The scheduling advisor then returns the host on which the task should be scheduled, as well as an estimate of its running time. The application can then use this information to schedule the task. The interface is as follows:

```
int RTAdviseTask(RTSchedulingAdvisorRequest       &req,
                 RTSchedulingAdvisorResponse       &resp);

struct RTSchedulingAdvisorRequest {
    double    tnom;
    double    slack;
    double    conf;
    Host      hosts[];
}

struct RTSchedulingAdvisorResponse {
    double                              tnom;
    double                              slack;
    double                              conf;
    Host                                host;
    RunningTimePredictionResponse       runningtime;
}
```

RTSchedulingAdvisorRequest expresses the scheduling problem: Choose a host from hosts such that a task with nominal running time tnom (t_{nom}), if started now, will complete in time $(1 + slack)t_{nom}$ or less with confidence conf. The real-time scheduling advisor's response consists of a copy of the request's t_{nom}, $slack$, and $conf$ values, the selected host, and an estimate of the task's running time, expressed as described in the previous chapter.

It is important to note that the scheduling problem may not have a solution because of a (predicted) lack of resources. If this is the case, the advisor will select the host which minimizes the running time of the task. It is the application's responsibility to verify, by using the runningtime field, whether the task is predicted to meet its deadline or not.

6.2 Implementing the interface

The implementation of the real-time scheduling advisor relies heavily on the running time advisor described in the previous chapter. Because the running time advisor does most of the work, the implementation of the real-time scheduling advisor's RTAdviseTask call is simple:

1. Construct a RunningTimePredictionRequest from the confidence level and nominal time in the RTSchedulingAdvisorRequest.

2. Use the PredictRunningTime interface to predict the running time (RunningTimePredictionResponse) for the task on each of the hosts. Note that this step can be done in parallel, since the majority of the computation happens in the host load prediction systems running on the individual hosts.

3. Find the subset of the hosts whose upper bound on the confidence interval for the running time of the task is less than the deadline, or $t_{ub} \leq (1 + slack)t_{nom}$. We refer to these as the *possible hosts*, because they are the hosts on which it is possible to meet the deadline with the desired confidence.

4. If there are no possible hosts add the host with the minimum expected running time (t_{exp}) to the set of possible hosts. At the end of this step, the number of possible hosts is always at least one.

5. Select a host at random from the set of possible hosts and return it and its corresponding `RunningTimePredictionResponse` to the caller via the `RTSchedulingAdvisorResponse`.

The advisor attempts to select a host which meets the user's goal of having the task meet its deadline while also attempting to limit the amount of contention with other advisors. In situations where most hosts have little load, the set of possible hosts will be large and it will be unlikely that two advisors choose the same host from it. When loads increase, the set of possible hosts will shrink and conflict will become more likely. As load increases further, the number of possible hosts will shrink to one and contention will become even more likely. We are able to introduce this randomness into the scheduler because the application expresses the slack it can tolerate. Simply put, there may be several hosts on which the application's deadline can be met, but only one on which the running time of the task is minimized.

As load increases further, the number of possible hosts will shrink to one, and, eventually, the advisor will also begin reporting to the application that the deadline cannot be met on that host. At this point, we expect that the application will begin to back off, either by increasing slack, or by changing the amount of work it wants to do.

The number of possible hosts depends not only on the load in the system and its variability, but also on how well the host load prediction systems can predict it, and how well the running time advisor can transform these predictions into confidence intervals for the running time. Better host load predictors and better modeling in the running time advisor will increase the likelihood that a host is marked possible, which will increase the number of deadlines that are met and lower the amount of contention between different real-time scheduling advisors.

6.3 Performance metrics

Evaluating the real-time scheduling advisor is considerably more complex than evaluating the running time advisor because many measures of its performance depend strongly on properties of the set of hosts (the scenario), and on properties of the application requests. The most intuitive measure of performance, the probability that a deadline will be met by following the advisor's advice, which we can measure as the fraction of some set of randomized tasks that meet their deadlines, has a number of problems. Consider the following examples. If a request has a very large slack, then almost any scheduling decision will result in the deadline being met. If two hosts have drastically different long term mean loads and little variability, then even a low quality load predictor will probably suffice to meet the deadline. If slack is very low and all of the hosts are heavily loaded, then no scheduler may be able to meet the deadline.

Of course, if we knew the *optimal* fraction of deadlines that could be met, we could compare against that value. As we discussed in Section 5.3, using a simulation approach to this problem would require a model to estimate running times on each of the hosts, but that model is a part of the system we are evaluating. This led us to a measurement-based approach. In this approach, we could conceivably submit each task to all of the available hosts and then determine the optimal fraction. However, this would perturb hosts that would not have been perturbed if an application were running the tasks, and thus the answer would be different.

Simply put, for n hosts, the evaluation system would put n times as much load on the hosts as an application would.

Another possibility is to use our load playback system to, for each task, reconstruct the environment and individually test each host. However, the environment depends not only on the background load, but also on the load placed on the system by previous tasks, thus we would have to reconstruct every possible set of task mappings, which grows exponentially in the number of tasks. Even if we were willing to ignore the effect of previous tasks, we would increase the time to run the evaluation by at least a factor of the number of hosts in the scenario. Currently, even with running the task only once, the evaluation requires about 1.5 days per scenario, during which time the n hosts are dedicated. Increasing this considerably is not practical at this time. Because of these complexities, we decided to introduce two additional metrics.

In addition to suggesting which host is most appropriate for running the task, the real-time scheduling advisor also provides a prediction of the running time of the task on that host. This prediction is expressed as a confidence interval, as discussed in the previous chapter. If the upper bound of that interval is greater than the deadline ($(1 + slack)t_{nom}$), then the advisor is informing the application that although this is the recommended host for the task, the advisor does not believe that the task will meet the deadline with the confidence the user requested. If the task then does not meet the deadline, is it the fault of the real-time advisor or the application? After all, the use of the real-time advisor has given the application the opportunity to change the task's compute requirements or slack such that the deadline can be met.

In order to capture failures to meet deadlines that are attributable solely to mistakes by the advisor, we also measure the fraction of deadlines met when the advisor believes it is possible. Ideally, this number should be identical to the confidence level requested by the user. Notice that it does not count situations where the advisor erroneously believes the deadline can not be met when it can. To measure this error would require being able to determine whether an individual testcase's deadline could have been met, which is difficult for exactly the same reasons that finding the optimal fractions of deadlines that could be met is difficult.

If multiple real-time scheduling advisors are active on the same distributed environment, contention is a concern. If all of the advisors target the same host—the one with minimum load, for example—then *all* of their tasks are more likely to miss their deadlines. Furthermore, if the schedulers then notice the vastly increased load on the targeted host, they may move, en masse, to target one other host. This sort of synchronized behavior would result in uniformly bad performance, even when plenty of resources are available to meet the tasks' deadlines. Randomness is a powerful way to break synchronization, and our implementation (Section 6.2) introduces randomness in its recommendations. That degree of that randomness depends on the number of hosts on which the deadline could be met. Thus, we will include the average number of possible hosts as a metric. Notice that a purely random scheduler is optimal with regard to this metric, while a scheduler based on minimizing expected time or host load is pessimal. Prediction-based schedulers have the potential to operate between these extremes.

The performance metrics that we shall use in this chapter are summarized below.

- **Fraction of deadlines met** : The fraction of testcases whose deadlines were met.

- **Fraction of deadlines met when possible** : The fraction of testcases where the deadline was met given that the predictor's estimate suggested that the deadline would be met.

- **Number of possible hosts** : The average, over the testcases, of the number of hosts on which task deadlines could be met.

6.4   Scheduling strategies

A scheduling strategy is an algorithm for deciding which host is most appropriate for running a task. Section 6.2 described a strategy based on the host load prediction-based running time advisor and randomness. The primary variable in such a system is which host load predictor is used. In keeping with the evaluations of host load prediction systems in Chapter 4 and of the running time advisor in Chapter 5, we studied the performance of scheduling strategies based on the MEAN, LAST, and AR(16) predictors. These running time advisor-based systems can tell the application not only which host to use, but also whether the task is likely to meet its deadline or not. Furthermore, they introduce randomness into their advice, which reduces the probability of contention between different scheduling advisors.

Simpler, but more limited scheduling strategies are also possible. We would like to assure ourselves that the additional complexity of the prediction-based strategies does indeed provide benefits over simpler approaches. To this end, we studied two additional scheduling strategies: RANDOM and MEASURE. The RANDOM strategy simply recommends a randomly selected host. This approach requires absolutely no infrastructure such as host load measurement or prediction systems and has no overhead. Furthermore, there is little chance of contention among RANDOM-based advisors. The MEASURE strategy measures the current load on each of the available hosts and then selects the host with the minimum load. Obviously, it is very prone to contention. MEASURE uses a host load sensor running on each of the hosts which introduces overhead. As we showed in Chapter 2, the overhead of a full-fledged host load prediction system based on an appropriate prediction model such as AR(16) is, in absolute terms, only marginally higher than that of a host load sensor. The benefit purchased with this miniscule extra overhead is the ability not only to choose an appropriate host, but also to predict what the actual running time will be on that host and thus whether the deadline is likely to be met or not.

In summary, the scheduling strategies that we evaluated are:

- **RANDOM**: Assigns the task to a randomly selected host. This strategy requires no external infrastructure, has zero overhead, and clearly provides the highest number of possible hosts. The fraction of deadlines met when possible metric is identical to the fraction of deadlines met metric because RANDOM has no way to tell if the host it chooses can actually meet the deadline.

- **MEASURE**: Assigns the task to the host with the lowest load. The number of possible hosts metric is almost always one. As with RANDOM, the fraction of deadlines met when possible metric is identical to the fraction of deadlines met metric.

- **MEAN** : The strategy described in Section 6.2 using MEAN as the host load predictor.

- **LAST** : The strategy described in Section 6.2 using LAST as the host load predictor.

- **AR(16)** : The strategy described in Section 6.2 using AR(16) as the host load predictor. AR(16) is the preferred host load predictor.

6.5   Modeling to gain intuition

Evaluating a prediction-based real-time scheduling advisor is a complex endeavor, because the performance of such systems depends not only on the quality of its predictions, but also on various properties of the set of hosts (the scenario) to which it assigns tasks and on properties of the requests the application makes. These dependencies are often counter-intuitive.

To gain an intuition to guide us in interpreting the empirical evaluation of our system (Section 6.6), we developed a simple analytic model of prediction-based real-time scheduling. The model shows how the

scheduling feasibility, which is the probability that there exists a host where a task can meet its deadline, and the *predictor sensitivity*, which is the probability that a deadline will be missed because of prediction error, depend on the number of hosts in the scenario, the mean slowdown of the hosts, the variability of slowdown across the hosts, the variability of slowdown over time for each host, the slack of the scheduling request, and the prediction error.

For a given scenario, there is a *critical slack* which is related to the mean slowdown. Intuitively, as the number of hosts in the scenario grows, the scheduling feasibility rapidly increases. Similarly, as slack increases, scheduling feasibility grows. The effect of both interhost and intrahost slowdown variability, however, depends on the relationship of the slack with the critical slack. At slack levels greater than the critical slack, increases in variability decrease the scheduling feasibility. However, for slack levels less than the critical slack, increases in variability lead to increases in scheduling feasibility.

The predictor sensitivity is independent of the number of hosts in the scenario for prediction-based strategies such as ours, and it decreases as slack and variability increase. The critical slack is also a turning point for the predictor sensitivity. Generally, below the critical slack, predictor sensitivity becomes extreme for low variability situations. However, tasks such as these are incredibly rare. Most tasks for which prediction sensitivity is high are those where the slack is near the critical slack and where variability is low. The 4LS scenario described in Section 6.6.4, for slacks from 0.75 to 1.0, is an example of this kind of situation.

6.5.1 Scheduling feasibility and predictor sensitivity

Consider scheduling a task of nominal time t_{nom} seconds on a pool of n hosts to maximize the probability that it will finish in $(1 + slack)t_{nom}$ seconds or less, $slack \geq 0$. If executed on host i, the task will encounter a slowdown S_i such that its execution time, $t_{exec_i} = S_i t_{nom}$. If $t_{exec_i} \leq (1 + slack)t_{nom}$, or, equivalently, $S_i \leq (1 + slack)$, the deadline has been met.

Because the slowdown that the task encounters depends on the host on which it is executed as well as the time when it is executed, it is impossible to say deterministically whether or not a specific task's deadline can be met. Instead, we shall consider the probability that the deadline is met. The S_is are the random variables whose distributions will affect this probability. We shall refer to this probability as the *scheduling feasibility* of the scheduling problem. If the scheduling feasibility is high, then it is easy to find a host on which the task will meet its deadline.

Intuitively, we would expect the scheduling feasibility to increase (scheduling the task become easier) as *slack* is increased and also as the number of hosts is increased. In both cases the number of hosts where the deadline could be met increases. It important to note that the S_i have variability. Each S_i changes over time. We would expect that the less dynamic each specific S_i is, the more feasible the scheduling problem is. For low levels of slack, the result should be opposite, with increases in variability increasing the scheduling feasibility. In addition to each host's individual variability, the variability of the group of hosts as a whole also should affect the scheduling feasibility. Counter-intuitively, when slack is tight, increased group dynamicity can actually increase the feasibility. For example, it is easier to schedule in a tight slack situation when one S_i is consistently lower than others. Some authors have referred to the benefit of "unbalancing" load in traditional distributed real-time systems [18].

Prediction enables the selection of an appropriate host—one where the deadline can be expected to be met. The predictor returns an estimated execution time (and confidence interval) for each of the hosts, $\hat{t}_{exec_i}$. If this value is less than the deadline, then we may chose host i to execute the task. If we do and it turns out that $t_{exec_i} > (1 + slack)t_{nom}$, then the deadline has been missed due to predictor error. The probability that this occurs is *predictor sensitivity* of the scenario.

An analytic model of prediction-based real-time scheduling should allow us to understand how scheduling feasibility and predictor sensitivity depend on the properties of the hosts on which we will schedule

and the quality of the predictions provided by the prediction system. Intuitively, prediction-based real-time scheduling is interesting in scenarios where the scheduling feasibility is reasonably high, and the choice of predictor matters most in scenarios where the predictor sensitivity is high.

6.5.2   Analytic model

In our model, we treat S_i as a random variable. The variability of S_i is attributable to two factors: the *interhost variability* of slowdown and the *intrahost variability* of slowdown. To capture interhost and intrahost variability, we write $S_i = X_i + Y_i$, and thus

$$t_{exec_i} = (X_i + Y_i)t_{nom} \tag{6.1}$$

where the random variable X_i represents the contribution of interhost variability and the random variable Y_i represents the contribution of intrahost variability.

Predictions introduce an additional source of variability: prediction error. We model this as an additional random variable, Z_i, and have

$$\hat{t}_{exec_i} = \hat{S}_i t_{nom} = (X_i + Y_i + Z_i)t_{nom} \tag{6.2}$$

We assume that X_i, Y_i, and Z_i are independent of each other, and thus the S_is are also independent of each other. We also assume that all random variables are IID. Finally, in the interest of tractability, we shall assume that they are all normally distributed:

$$
\begin{aligned}
X_i &\sim N(\mu_s, \sigma_s^2) \\
Y_i &\sim N(0, \sigma_{sl_i}^2) \\
Z_i &\sim N(0, \sigma_{pred_i}^2) \\
S_i &\sim N(\mu_s, \sigma_s^2 + \sigma_{sl_i}^2)
\end{aligned}
\tag{6.3}
$$

μ_s represents the *mean slowdown*, the average slowdown encountered by tasks on this set of hosts. For example, on a 2 host system where host A has a mean load of 1 and host B has a mean load of 3, $\mu_s = ((1+1) + (1+3))/2 = 3$. σ_s^2 represents the interhost variability. In the 2 host example, $\sigma_s^2 = (((1+1) - \mu_s)^2 + ((1+3) - \mu_s)^2)/1 = 2$. $\sigma_{sl_i}^2$ represents the intrahost variability on host i, while $\sigma_{pred_i}^2$ represents the prediction error variability on host i. To simplify the analysis later, we shall assume all of the hosts have the same intrahost variability and prediction error, thus we have $\sigma_{sl_1}^2 = \sigma_{sl_2}^2 = \ldots = \sigma_{sl_n}^2 = \sigma_{sl}^2$ and $\sigma_{pred_1}^2 = \sigma_{pred_2}^2 = \ldots = \sigma_{pred_n}^2 = \sigma_{pred}^2$.

It is important to point out that these are quite drastic assumptions. However, the purpose behind our modeling effort is not quantitative, but qualitative. We want to identify the attributes of interesting scenarios and explain the performance results of particular scenarios by appealing to particular attributes of the model. Our assumptions depart from reality in the following ways. As we discovered in Chapters 4 and 5, S_i is actually correlated over time and thus predictable. However, we are modeling that predictability by making $\hat{t}_{exec_i}$ identical to t_{exec} except for an additional prediction error term $Z_i t_{nom}$. As we showed in Chapter 5, the prediction error does indeed grow somewhat linearly with t_{nom}. A more critical assumption is that the interhost variability and the prediction error are each modeled as a single random variable. There is nothing in our analysis that requires this other than tractability. However, it is a weak point. The normality assumption for prediction error is reasonable for reasonably large t_{nom} since multiple (non-normal) errors are averaged.

Using this model, we can now express the scheduling feasibility as

$$P[min_{i=1}^n t_{exec_i} \le t_{nom}(1 + slack)]$$

or

$$P[min_{i=1}^{n}\{(X_i - \mu_s) + Y_i\} \leq (1 + slack - \mu_s)] \tag{6.4}$$

For a single host, $X_1 = \mu_s$, and we have a scheduling difficulty of

$$P[Y_1 \leq (1 + slack - \mu_s)] \tag{6.5}$$

which is simply the CDF of $N(0, \sigma_{sl}^2)$ evaluated at $1 + slack - \mu_s$. $\mu_s - 1$ is the *critical slack* for the scenario—it is smallest level of slack for which 50% of the tasks will meet their deadlines. For slacks less than the critical slack, fewer tasks will meet their deadlines, and for higher slacks, more tasks will meet their deadlines.

The predictor sensitivity is considerably more complex. A major issue is what the scheduling strategy is—ie, which host will be chosen when more than one can meet the deadline. It is difficult to model the precise policy that our prediction-based strategies use (Section 6.2), but we shall look at two variants and show that they are similar. The basis of our policy is to find the set of hosts where the predicted running time is less than the deadline and then choose randomly from among those hosts. For this strategy, the predictor sensitivity is independent of the number of hosts because each host can be treated individually, so we can simply look at one host, say host 1:

$$P[t_{exec_1} > t_{nom}(1 + slack) \mid \hat{t}_{exec_1} \leq t_{nom}(1 + slack)]$$

or

$$P[(X_1 - \mu_s) + Y_1 > (1 + slack - \mu_s) \mid (X_1 - \mu_s) + Y_1 + Z_1 \leq (1 + slack - \mu_s)] \tag{6.6}$$

For one host, $X_1 = \mu_s$ and we have a prediction sensitivity of

$$P[Y_1 > (1 + slack - \mu_s) \mid Y_1 + Z_1 \leq (1 + slack - \mu_s)] \tag{6.7}$$

In our scheduler, if the set of hosts for which the predicted running time is less than the deadline is empty, an aggressive choice of host is made, namely the one with the minimum predicted running time. Now we'll consider a similar aggressive scheduler, where the host with minimum predicted running time is always returned. The predictor sensitivity for that scheduler is

$$P[t_{exec_{argmin_{i=1}^{n} \hat{t}_{exec_i}}} > t_{nom}(1 + slack) \mid min_{i=1}^{n} \hat{t}_{exec_i} \leq t_{nom}(1 + slack)]$$

The meaning of this expression is the following. Suppose we look at the predicted running time of all the hosts and pick host where that time in minimum. Suppose that the time is less than the deadline. What now is the probability that the deadline is missed? Expanding the equation we get

$$P[(X_{argmin_{i=1}^{n}(X_i+Y_i+Z_i)} - \mu_s) + Y_{argmin_{i=1}^{n}(X_i+Y_i+Z_i)} > (1 + slack - \mu_s)$$
$$\mid min_{i=1}^{n}(X_i + Y_i + Z_i) - \mu_s \leq (1 + slack - \mu_s)] \tag{6.8}$$

Unfortunately, for $n > 1$, we found this probability to be too difficult to get into a form to be computed. For a particular n, the decomposition into cases is automatic, but the integrals required for each case contain so many dimensions ($2n$ dimensions) that it is extremely difficult to determine what their bounds should be. For a single host, For a single host, $X_1 = \mu_s$ and this simplifies to

$$P[Y_1 > (1 + slack - \mu_s) \mid (Y_1 + Z_1) \leq (1 + slack - \mu_s)] \tag{6.9}$$

Notice that this is identical to Equation 6.7. Furthermore, notice that the non-aggressive scheduler's predictor sensitivity with multiple hosts (Equation 6.6) is of similar form. Essentially all three of these equations are of the form

$$P[W_1 > (1 + slack - \mu_s) \mid (W_1 + Z_1) \leq (1 + slack - \mu_s)] \tag{6.10}$$

where W_1 and Z_1 are both normally distributed random variables. W_1 has mean μ_s and some variance that depends on the intrahost and interhost variability and Z_1 is as before. Later we will examine this expression to characterize the predictor sensitivity of the non-aggressive scheduler for multiple hosts and the aggressive scheduler for a single host.

6.5.3 Intuition from the model

The model described in Section 6.5.2 is unfortunately not analytically tractable. However, we can use symbolic math software to study the relationship between the scheduling feasibility and predictor sensitivity and the parameters of the model: n, μ_s, σ_s, σ_{sl}, and σ_{pred}. In particular, we will study Equation 6.4 (scheduling feasibility) and Equation 6.10 (prediction sensitivity).

(a) 1 Host, $\sigma_s = 0$ (b) 2 Hosts, $\sigma_s = 0$ (c) 4 Hosts, $\sigma_s = 0$

(d) 1 Host, $\sigma_s = 0.5$ (e) 2 Hosts, $\sigma_s = 0.5$ (f) 4 Hosts, $\sigma_s = 0.5$

(g) 1 Host, $\sigma_s = 1.0$ (h) 2 Hosts, $\sigma_s = 1.0$ (i) 4 Hosts $\sigma_s = 1.0$

Figure 6.1: Scheduling feasibility as a function of number of hosts n, interhost variability σ_s, intrahost variability σ_s, intrahost variability σ_{sl} and $slack$ for mean slowdown $\mu_s = 2$ for slack values greater than the critical slack.

Scheduling feasibility

Figure 6.1 shows how scheduling feasibility varies according to the number of hosts n, the interhost variability σ_s, and the intrahost variability σ_{sl}. The scheduling feasibility is plotted in separate graphs for $n = 1, 2, 4$ $\times$ $\sigma_s = 0.0, 0.5, 1.0$. The number of hosts increases from left to right (eg, (a), (b), (c)) while the interhost variability increases from top to bottom (eg, (a), (d), (g)). The mean slowdown μ_s is set to 2.0, which corresponds to a mean load of 1.0. Each individual graph plots the scheduling feasibility as a function of the intrahost variability σ_{sl} for $\sigma_{sl} = 0.1$ to 1.0. Each curve corresponds to a particular slack, which ranges from 1.1 (the bottom curve on each graph) to 2.0 (the top curve on each graph). All of the slack values are greater than the critical slack for this situation ($slack \geq \mu_s - 1 = 1$).

Increasing the number of hosts the scheduler can schedule on dramatically increases the scheduling feasibility. The benefit of increasing the number of hosts increases with the the intrahost variability. The more dynamic the individual hosts are, the more of them that are desirable. As we can see from Figure 6.1(c), with four hosts, it is still > 90 % probable that the deadline can be met even with very dynamic hosts ($\sigma_{sl} = 1$) and very little slack (1.1). With one host (Figure 6.1(a)), that probability is only 60 %.

Increasing the slack also increases the scheduling feasibility. The benefit increases as the intrahost variability decreases. For example, in the single host scenario (Figure 6.1(a)), with an intrahost variability $\sigma_{sl} = 0.1$, increasing the slack from 1.1 to 1.2 increases the scheduling feasibility from about 85 % to about 97 %, while the same increase at $\sigma_{sl} = 1.0$ only increases the scheduling feasibility from 60 % to 65 %.

As the number of hosts increases, the benefit of increasing slack becomes much less important. With one host (Figure 6.1(a)), increasing the slack from 1.1 to 2.0 increases the scheduling feasibility from 60 % to about 85 %. The gain from the corresponding increase in the 4 host case (Figure 6.1(c)) is negligible.

Increasing the interhost variability σ_s has two effects. First, it decreases the scheduling feasibility overall. This is simply because we are introducing additional variability into the system. The more interesting effect is that that it flattens all of the curves. This is due also the increased variability.
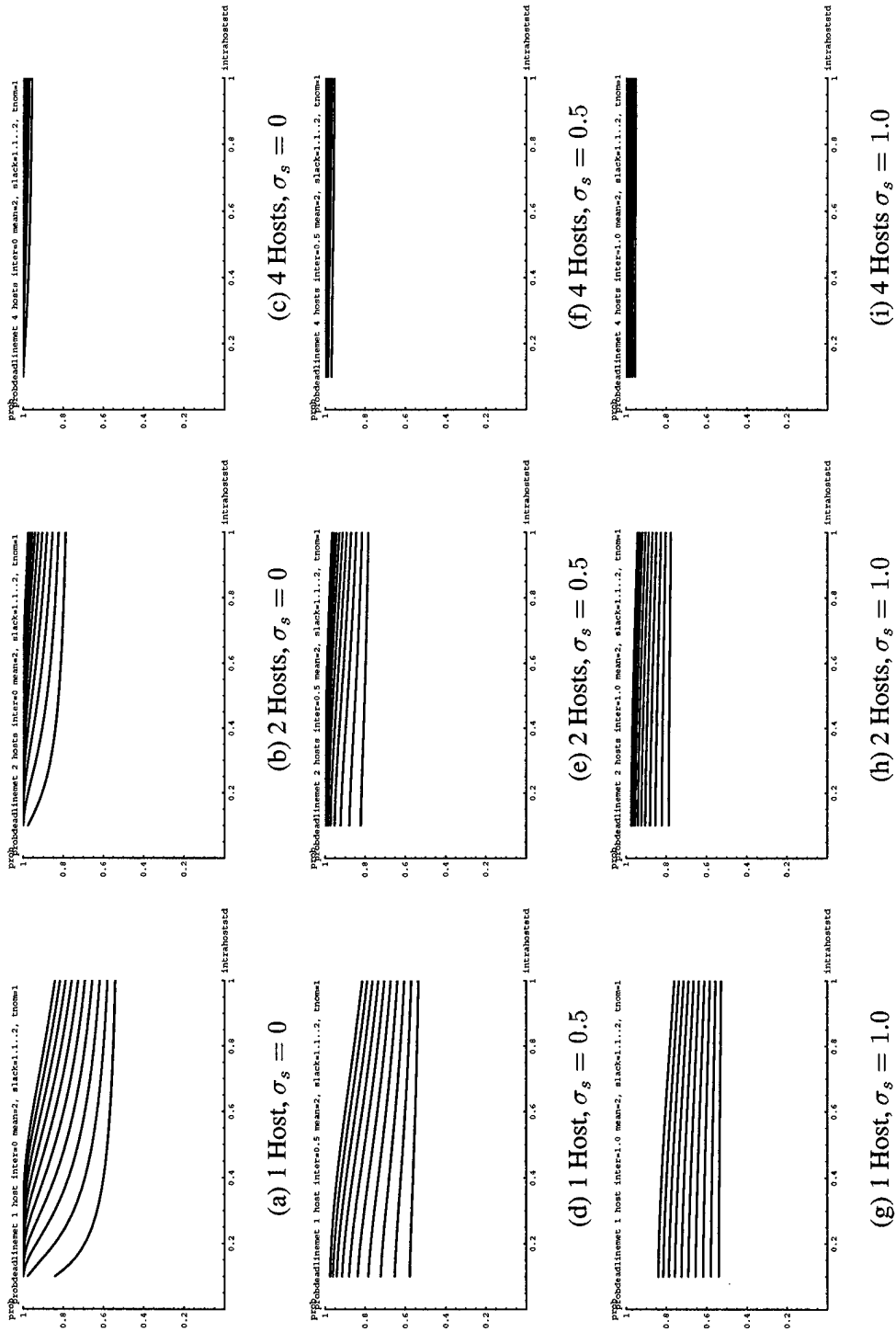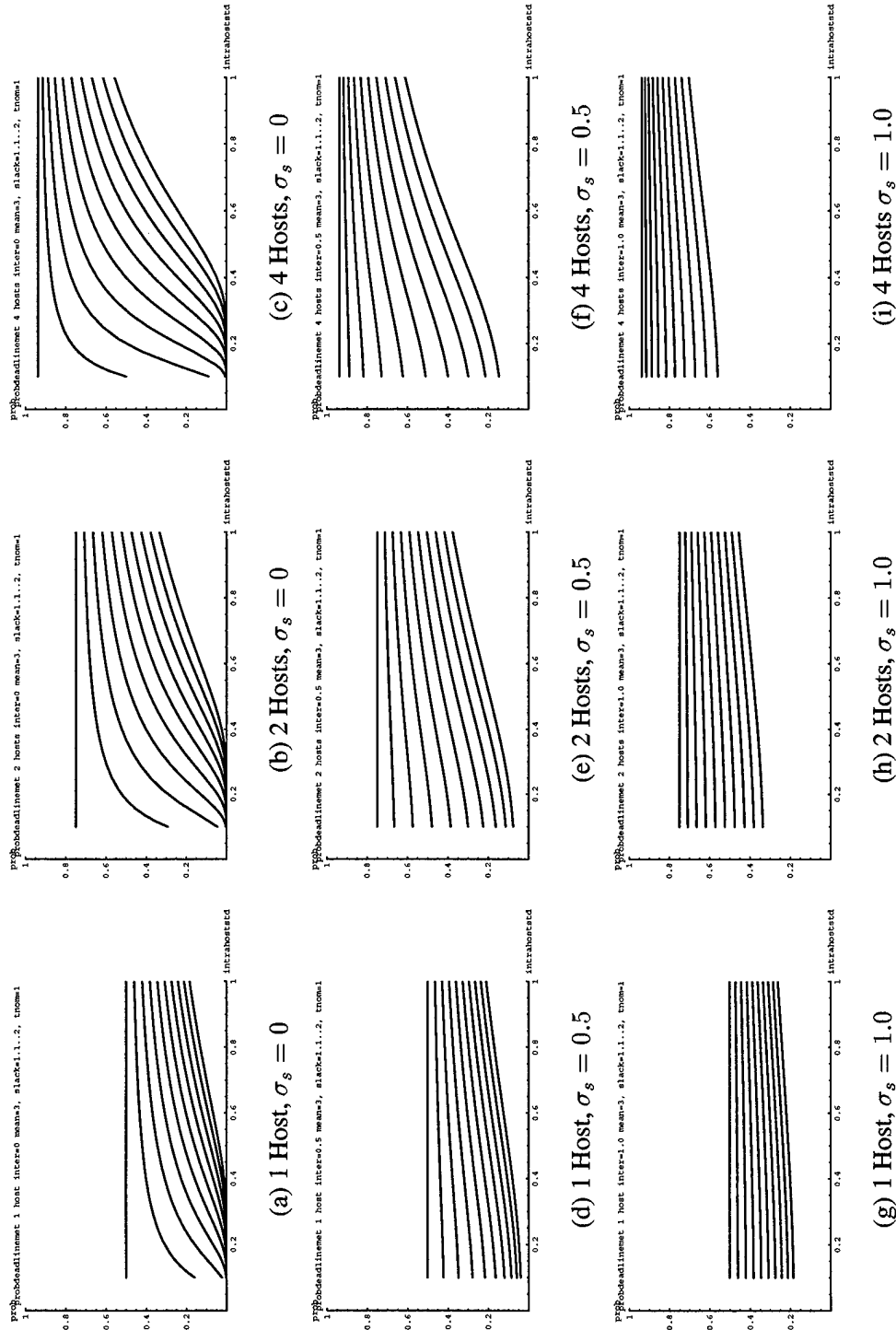
Figure 6.2: Scheduling feasibility as a function of number of hosts n, interhost variability σ_s, intrahost variability σ_{sl} and $slack$ for mean slowdown $\mu_s = 3$ for slack values less than the critical slack.

When slacks below the critical slack are considered, the scheduling feasibility's relationship with the scenario parameters and slack changes rather drastically. The parameters and explanation for Figure 6.2 are identical to those of Figure 6.1, which was discussed earlier. The only difference is that the mean slowdown μ_s has been increased to 3.0. This makes the critical slack 2.0 and now all of the slack levels considered in the figure are less than the critical slack.

In some respects, the behavior of the scheduling feasibility is the same as before. For example, it still grows with the number of available hosts. More hosts seem to always be a good thing. Also as before, the scheduling feasibility increases as the slack is increased—for each graph in the figure, the slack curves represent increasing slack from bottom to top.

The major difference between being above or below the critical slack is the effect of variability. Above the critical slack (Figure 6.1), additional variability,whether it is due to increasing interhost or intrahost variance or both, decreases the scheduling feasibility . In contrast, below the critical slack, increases in variability from either source increases the scheduling feasibility. To understand this counter-intuitive result, consider a one host case. Suppose that $\mu_s = 2$ and $\sigma_{sl} = 1$. The slowdown that a task will see is then normally distributed around 2.0 with a variance 1.0. Now consider a slack of 2, which is above the critical slack by one standard deviation. The probability that the deadline can be met with this slack is then the CDF of the slowdown evaluated at 3, which is considerably to the right of the normal distribution's maximum. Furthermore, it is is one standard deviation to the right, and thus the feasibility is about 84%. Suppose the variance doubles. The normal is now much flatter and more probability has moved to the right of 3. In fact, the feasibility has declined to about 76%. Clearly the increase in variability has hurt the feasibility. Now repeat the above steps with a slack of zero, which is one standard deviation to below of the original normal. The scheduling feasibility is the CDF of the normal evaluated at 1.0, which is about 16%—it is unlikely that the deadline can be met. If we increase the variance as before, the normal flattens and now more probability is to the *left* of 1.0, which raises the scheduling feasibility to about 24%.

Figure 6.3: Predictor sensitivity as a function of *slack*, intrahost and interhost variance $(\sigma_s + \sigma_{sl})$, and prediction error σ_{pred} on 1 host for $\mu_s = 2$ for slack values greater than the critical slack.

Predictor sensitivity

Figure 6.3 shows how the predictor sensitivity varies with the slack, prediction error, and intra/interhost variability for slacks that are greater than the critical slack. The figure plots Equation 6.10 with $\mu_s =$ 2. In the figure, each graph shows a different slack value, ranging from 1.0 (Figure 6.3(a)) through 2.0 (Figure 6.3(b)). Each individual graph plots the predictor sensitivity as a function of the prediction error variance (σ_{pred}) for a family of different values for the sum of intrahost and interhost variability. For a particular graph the curves are arranged in decreasing order of variability from bottom to top.

Not surprisingly, we see that the predictor sensitivity declines as the slack increases. As this happens, more and more prediction errors are moot simply because there is a greater "cushion" of time to cover the slowdown variability. For large slack levels, it doesn't really matter what predictor is used. However, as slack is decreased and comes close to the critical slack, the predictor sensitivity sky-rockets. Consider what happens at the critical slack on a single host. At this point the scheduling feasibility is precisely 0.5—there is a 50% chance of meeting the deadline. Suppose the intrahost variability is small compared to the prediction error variance. Then, if the predictor claims that a deadline can be met, the chance that it is wrong can actually outweigh the chance that the deadline can be met. At this point at the critical slack, there are lots of tasks that could meet their deadlines (50% of them!), but only if the prediction error variance is considerably lower than the variance in slowdowns.

As the interhost and intrahost slowdown variance increases, the predictor sensitivity decreases. The closer we are to the critical slack, the more quickly it decreases with increasing slowdown variance. Essentially, as the slowdown variance increases, the chance that a deadline is missed due to a lack of resources is increasing while the chance that it is missed due to a miss-prediction is decreasing.
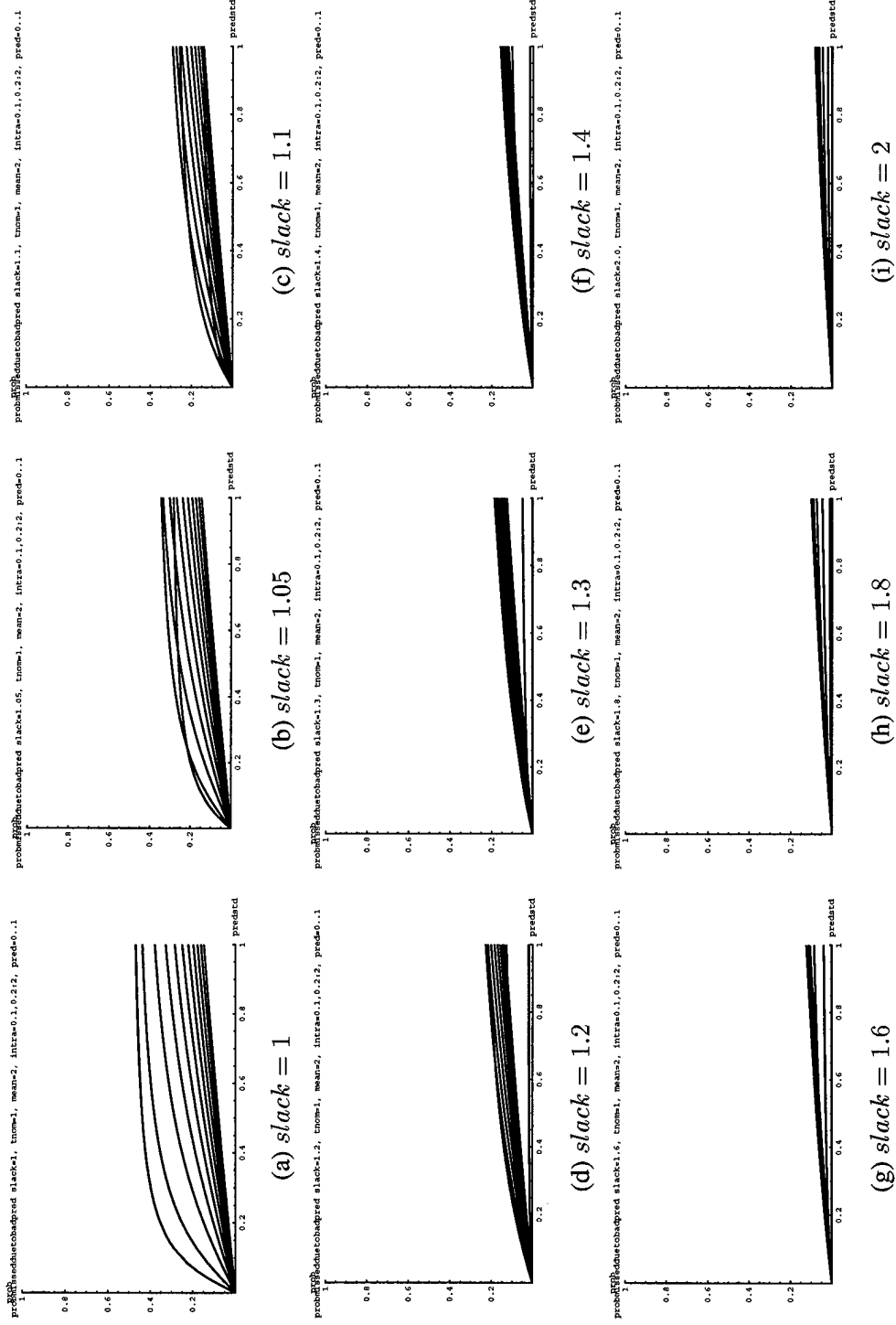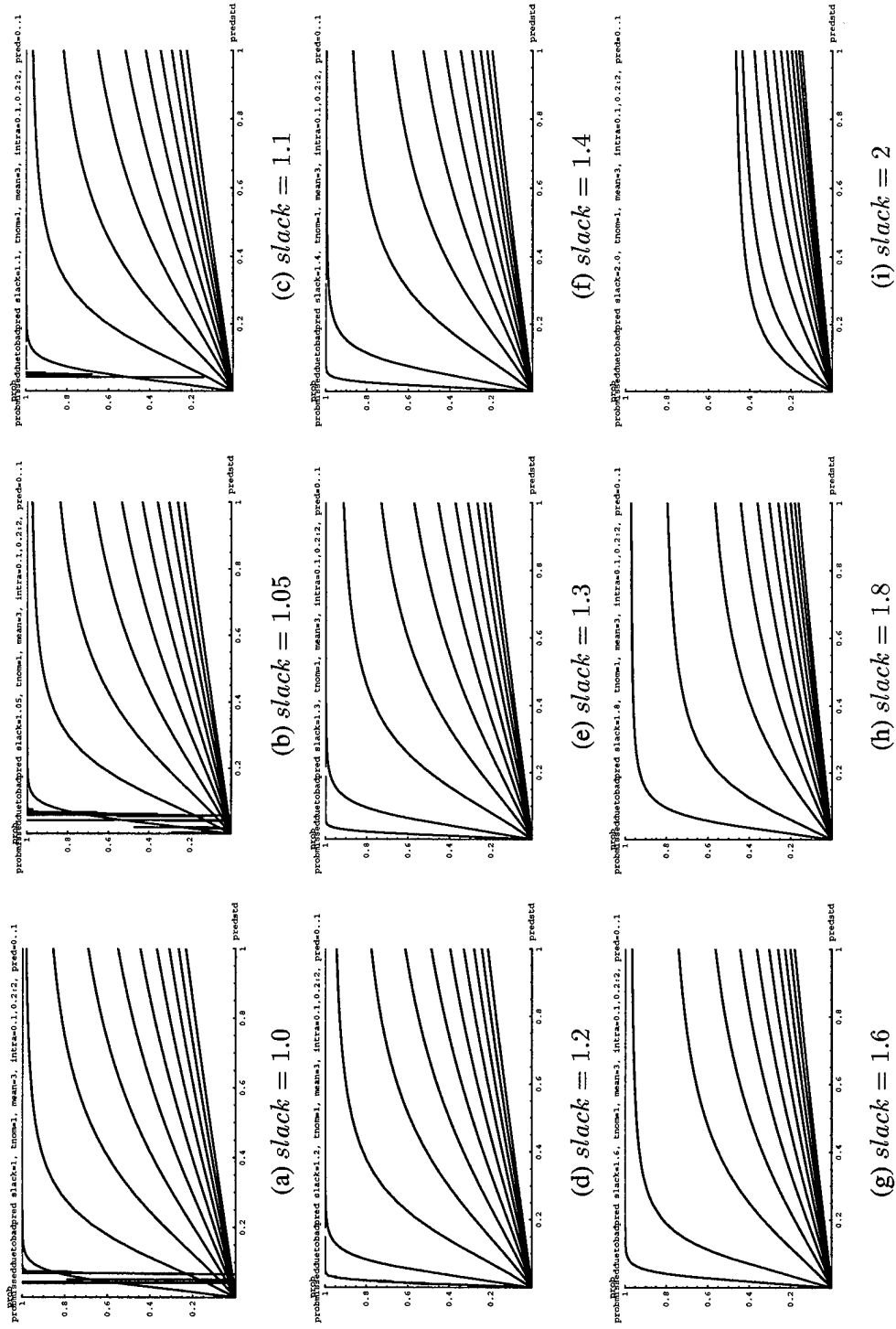
Figure 6.4: Predictor sensitivity as a function of *slack*, intrahost and interhost variance $(\sigma_s + \sigma_{sl})$, and prediction error σ_{pred} on 1 host for $\mu_s = 3$ for slack values less than the critical slack.

Unlike the scheduling feasibility, the predictor sensitivity's relationship to slack, intrahost and interhost variance, and prediction error remains the same as we transition from slack values greater than the critical slack to slack values less than the critical slack. Figure 6.4 was computed using the same parameters as Figure 6.3, except that the mean slowdown μ_s was set to 3.0. This makes all the slack values less than the critical slack.

If we look at decreasing slack values from 2.0 (Figure 6.4(i)) down to 1.0 (Figure 6.4(a)), we can see that predictor sensitivity continues to grow with decreasing slack. However, the figures do not tell the whole story. The scheduling feasibility is also declining precipitously as as we decrease the slack (see Figure 6.2), so for the fewer and fewer tasks that can have their deadlines satisfied, the predictor is increasingly important. However, since the tasks are so very few, we would not expect this additional sensitivity to greatly affect the overall probability that a deadline can be met, even if we upgraded to a remarkably prescient predictor. As we saw earlier, on the other side of the critical slack, the predictor sensitivity rapidly drops as we increase slack, but the scheduling feasibility insures that more tasks can have their deadlines met and thus a better predictor will not significantly change the overall probability that a task will be met. The region where improving the predictor has the greatest impact is near the critical slack. This is precisely what we see in our evaluation of the 4LS scenario (Section 6.6.4).

6.6 Evaluation

We evaluated the five different strategies for implementing the real-time scheduling advisor by using an experimental infrastructure similar to that of the last chapter to run large numbers of testcases that were randomized with respect to their starting time, the slack level, the nominal time, and the strategy used. For every testcase, the advice proffered by the advisor was followed, the testcase's task was run on the suggested host, and the results logged. Using the testcases produced by applying this methodology, we then estimated the values for the three performance metrics under different constraints on the slack level and nominal time.

The performance metrics depend not only on these randomly selected parameters but also on the set of hosts on which the advisor can schedule tasks (the scenario) that the advisor faces. As we argued in Section 6.5, the properties of the scenario, along with the slack, should affect the scheduling feasibility and the predictor sensitivity. Using the statistics that best cluster the hosts (Chapter 3), we constructed several representative scenarios based on the August, 1997 traces. We also introduced two additional scenarios that seemed intuitively interesting. We evaluated the performance of the different scheduling strategies on each individual scenario as described above.

In the following, we will first give a detailed description of the experimental infrastructure, the methodology, and the scenarios we evaluated. Next, we will examine one of the scenarios, 4LS, in considerable detail. The main findings are summarized in the following three paragraphs.

The MEASURE and prediction-based strategies significantly outperform the RANDOM strategy at all slack levels and all nominal times in terms of the fraction of deadlines met. The AR(16) strategy is the best of the prediction-based strategies, it always performs at least as well as the MEASURE strategy, and considerably improves on its performance at "near-critical" levels of slack, where it is just feasible to meet deadlines and where predictor sensitivity should be highest according to model we developed in Section 6.5. The fraction of deadlines met declines for all the strategies as the nominal time of the task increases. Along this dimension, we see a similar relationship between RANDOM, MEASURE, and the prediction-based strategies: MEASURE is very preferable to RANDOM, and AR(16) improves on its performance in some cases, particularly at near-critical slacks.

Unlike the RANDOM and MEASURE strategies, the prediction-based strategies are able not only to choose an appropriate host for a task, but they can also tell the application whether they believe the deadline can be met on that host. This allows us to differentiate between deadlines that are missed because the strat-

egy incorrectly predicted there were sufficient resources and deadlines that are missed because there simply weren't sufficient resources. The fraction of deadlines met when possible metric is essentially the probability that a deadline will be met given that the predictor believes it can be met. In terms of this metric, for the 4LS scenario, the prediction-based strategies considerably outperform the MEASURE and RANDOM strategies, and the AR(16) strategy considerably outperforms the other prediction-based strategies. Furthermore, AR(16)'s performance is relatively independent of slack and nominal time, dipping only slightly at the critical slack level and for medium-sized tasks. Consider what this means. Suppose an application asks the advisor for a host on which its task will meet its deadline with 95% probability. If the advisor claims that the deadline can be met on the host it chooses, then the application can be quite certain that if it sends the task to that host, it will indeed meet its deadline with 95% probability.

The prediction-based strategies are able to convert excess slack, or temporarily low load, into randomness in their scheduling decisions. This randomness is useful because it can keep different advisors from synchronizing their choices, resulting in uniformly bad performance. Such a catastrophic synchronization could conceivably occur with the MEASURE strategy, for example. For the 4LS scenario, this randomness, as measured by the average number of possible hosts metric, increases with slack and decreases with nominal time. Even at low slack levels and long nominal times, however, the prediction-based strategies are able to introduce some randomness. All three prediction-based strategies produce similar results.

The overall conclusion about the 4LS scenario is that using the AR(16) strategy produces the best results in terms of all three metrics. It is clear that RANDOM is insufficient—a MEASURE strategy, at least, is indicated. As we showed in Chapter 2, AR(16) has little additional overhead over a MEASURE strategy, and, as we show here, it has a number of benefits:

- AR(16) can meet at least as many deadlines as MEASURE, and, at near-critical slack values, AR(16) can meet considerably more.

- Unlike MEASURE, AR(16) can assert that it thinks the deadline can be met by using the host it recommends, and the application can be extremely confident that the assertion will be true.

- Unlike MEASURE, AR(16) can introduce significant amounts of randomness into its scheduling decisions, which reduces the risk of catastrophic synchronization with other advisors.

The results for the other scenarios are, for the most part, similar to those of 4LS, and we present them in an abbreviated form that makes it easy to compare all of the scenarios, including 4LS. We show how the performance metrics for each scenario vary with slack, nominal time, and jointly with slack and nominal time. One scenario, 2CS, behaves differently from the others. On 2CS, RANDOM performs better than MEASURE, and the simpler prediction strategies perform better than the more complex ones (although AR(16) still performs at least as well as MEASURE). This odd-man-out argues, although mildly, for a multiple-expert system, where several prediction-based strategies run simultaneously, and the best-performing one is the one used.

The final part of our evaluation attempts to characterize advisor contention by having multiple advisors schedule tasks on a small, 2 host scenario. We don't see any occurrence of synchronization, and all of the advisors see roughly the same performance. This suggests that the combination of randomness introduced by the prediction-based strategies, as well as the unsynchronized task submission times (which are characteristic of multiple interactive applications) is sufficient to avoid this potential problem with our approach to resource prediction-based real-time scheduling advisors. Although the advisors are oblivious of each other, there is sufficient randomness in their actions that they rarely contend for long. Despite this, they are able to produce much better performance for the applications than a purely RANDOM strategy.

6.6.1   Experimental infrastructure

Our evaluation of the real-time scheduling advisor relies on essentially the same infrastructure as the evaluation of the running time advisor (Section 5.3). That infrastructure consisted of one recording host and one measurement host. We extend this to include more than one measurement host and have the recording host choose which host a task is sent to based on the real-time scheduling advisor.

The measurement hosts are reserved Alphastation 255s on a private network as before. Each host runs the same software as described in Section 5.3, but each plays back a different host load trace. The set of load traces being played back defines the scenario that is being studied.

The biggest change in the infrastructure is the behavior of the recording host. To evaluate the running time advisor, the recording host used that advisor's interface to predict the running time of a task on the measurement host, measured the actual running time by executing the task on the measurement host, and finally recorded the prediction and the measurement to a file. To evaluate the real-time scheduling advisor, the recording host used the interface of Section 6.1 to select the most appropriate host in the scenario, runs the task on that host and measures its running time, and records the selected host, the predicted running time, and the actual running time.

6.6.2   Methodology

To evaluate the real-time scheduling advisor given a particular scenario, we started up the infrastructure described in Section 6.6.1. Each of the measurement hosts was set to play back one of the load traces in the scenario. Each measurement host also ran a host load sensor at 1 Hz, three host load prediction systems (MEAN, LAST, and AR(16)). The systems were configured to fit to 300 measurements (5 minutes) and to refit themselves when the absolute error for a one-step-ahead prediction exceeded 0.01 or the average measured one-step-ahead mean squared error exceeded the estimated one-step-ahead mean squared error by more than 5%. The minimum interval between refits was 30 seconds, and maximum interval before the measured mean squared error was tested was 300 seconds. This setup was identical to that of Section 5.4.1, except that it ran on multiple measurement hosts.

The prediction and measurement software were permitted to quiesce for at least 600 seconds and then 8000 consecutive testcases were run on the recording host, each according to this procedure:

1. Wait for a delay interval, $t_{interval}$, selected from a uniform distribution from 5 to 15 seconds.

2. Get the current time, t_{now}.

3. Select the task's nominal time, t_{nom}, randomly from a uniform distribution from 100 ms to 10 seconds.

4. Select the task's slack, $slack$, randomly from a uniform distribution from 0 to 1.

5. Select a scheduling strategy randomly from among RANDOM, MEASURE, MEAN, LAST, and AR(16).

6. Depending on the scheduling strategy chosen in step 4, choose the target host for the task in one of the following ways:

   - RANDOM : Select one of the hosts in the scenario at random.

   - MEASURE : Request load measurements from each of the hosts in the scenario. Select a host at random from among those with the minimum load.

   - MEAN, LAST, AR(16) : Use the `RTAdviseTask` API's (Section 6.1) implementation (Section 6.2) to select the host using the chosen host load predictor.

7. Run the task on the spin server of the chosen host and retrieve its actual running time, t_{act}.

8. Record the timestamp t_{now}, the scheduling strategy used, the nominal time t_{nom}, the slack *slack*, the expected running time t_{exp}, the confidence interval $[t_{lb}, t_{ub}]$, and the number of possible hosts.

After all 8000 testcases were run, their records were imported into a database table corresponding to the scenario. In the case of the 4LS scenario (see below), we looked at slack values in the range zero to two and completed 16000 testcases. It takes approximately 36 hours to complete 8000 testcases. To evaluate the real-time scheduling advisor, the database is mined.

6.6.3 Scenarios

When invoked, the real-time scheduling advisor chooses which of the available hosts is most appropriate for running the task such that it meets its deadline. In part, the performance of the advisor depends on the properties of this group of these hosts (the scenario). To evaluate the advisor, it is important to consider several different scenarios that are likely to be seen in practice. We will now introduce the scenarios that we used in our evaluation.

For the most part, we based the construction of our scenarios on how our load traces are clustered by their statistics. We then added several additional scenarios based on other environments in which we could see real-time advisors operating. Our scenarios are also connected to the parameters of the model we developed in Section 6.5. Obviously, we did not attempt to explore the space of those parameters. Nonetheless, the scenarios (and hopefully our results!) can be interpreted in the framework of that explanatory model.

In our experimental environment, scenarios are represented by a group of host load traces. We used the August, 1997 set of traces. In clustering the traces by their statistics, the most significant statistics are the mean load and the mean epoch length. We classified each trace as having "low" or "high" mean load, and "small" or "large" mean epoch lengths. For most of the scenarios that we constructed, these formed the basis of the construction. The traces in a scenario could all be of low mean load, high mean load, or a "mixed" combination. Similarly, the scenario's traces could all be of small epochs, large epochs, or a mixed combination. This forms nine classes. Unfortunately, because we have no traces which are simultaneously of high mean load and long mean epoch length, only six of the classes were realizable. Of these six combinations, we will report on four particularly interesting ones here. These are:

- **4LS** : These four traces (axp0, axp4, axp5, and axp10) all exhibit high mean load and small epochs.

- **4SL** : These four traces (axp3, axp7, axp8, and axp9) all exhibit low mean load and large epochs.

- **4MM** : These four traces (axp0, axp4, axp7, and axp8) exhibit mixed load and epochs. axp0 and axp4 have high mean load and small epochs, while axp7 and axp8 have low mean load and large epochs.

- **5SS** : These five traces (axp1, axp2, axp6, axpfea, and axpfeb) exhibit low load and small epochs.

The two combinations that we do not report on here are the 4MS scenario (axp0, axp1, axp2, and axp4), which represents mixed loads and small epochs, and 4SM (axp1, axp2, axp7, and axp8), which represents low loads and mixed epochs. The results for these two scenarios are similar to those we report here. We also chose to add an additional scenario based on a compute server environment:

- **2CS** : These are the traces of the two compute server machines (mojave and sahara).

Finally, we added a scenario that we perceived as highly predictable in order to determine how multiple real-time advisors might interact. The thinking was that for a highly predictable scenario (one where the predictor sensitivity is low), performance degradation due to contention would be more noticeable. This scenario is

- **2MP** : Two of the more predictable traces (axp8, and axpfea).

In Section 6.5 we derived a simple analytic model for the real-time advisor in order to help us understand how the difficulty of finding an appropriate host (scheduling feasibility) and the importance of good predictions (predictor sensitivity) depend on the scenario, the prediction quality, and the slack. The model characterized the scenario by the number of hosts it contained (n), its mean slowdown (μ_s), its interhost variance in slowdown (σ_s^2) and its intrahost variance in slowdown (σ_{sl}^2).

In terms of that analytic model, our scenarios represent different combinations of the parameters n, μ_s, σ_s^2, and σ_{sl}^2. The number of hosts in the scenarios (n) range from two to five. As we discovered in Chapter 3, load variability is strongly correlated with mean load. Load variability, in turn, is the source of the intrahost variability in slowdown (σ_{sl}^2). The high load scenario 4LS represents a high mean slowdown μ_s because all of the hosts have high mean load. Correspondingly, the intrahost variability (σ_{sl}^2) of 4LS is also high. In contrast, the interhost variability (σ_s^2) is low because all the hosts have similar mean loads. The low load scenarios 4SL and 5SS have low μ_s, σ_s^2, and σ_{sl}^2. Because the correlation of mean load and the variance of load suggests that μ_s and σ_{sl}^2 are also correlated, scenarios with low μ_s coupled with high σ_{sl}^2 or high μ_s coupled with low σ_{sl}^2 are impossible to construct using our traces. The "mixed" scenario, 4MM has μ_s and σ_{sl}^2 levels between those of 4LS and 4SL/5SS, while it has considerable interhost variability (σ_s^2). The smaller scenarios 2CS and 2MP represent low μ_s, σ_s^2, and σ_{sl}^2 situations combined with small n.

6.6.4   4LS scenario in detail

The performance of the real-time scheduling advisor depends on the scenario, the nominal running time of the task, and the slack level. In this section, we present the 4LS scenario in detail. The overall behavior of the performance metrics with respect to the nominal time and the slack are similar for all the scenarios. This section allows us to illustrate that behavior in detail. 4LS is also interesting because in it we see the greatest differentiation between the different scheduling strategies. Unlike the other scenarios, however, the study of 4LS included slack values from 0 to 2, resulting in 16000 testcases. This was necessary because even with a slack value of 1, a large number of tasks did not meet their deadlines. Because we wanted to see how the different strategies performed in situations ranging from very low to very high scheduling feasibility, increasing the slack was necessary. In the subsequent sections we will show aggregated results for 4LS as well as the other scenarios, all for slacks ranging from 0 to 1.

Performance versus slack

Figure 6.5 illustrates how the performance metrics vary with slack for each of the scheduling strategies using the 4LS scenario. A point represents the interval of slack times between it and the preceding point. For example, the fraction of deadlines met for slacks from 0 to 0.25 using the LAST strategy is plotted as a disc at (0.25, 0.2) in Figure 6.5(a).

The fraction of deadlines met is a function of both the quality of the scheduling strategy and the scheduling feasibility. When slack is very low, we are in a regime with low scheduling feasibility where encountering a host on which a task can be successfully scheduled is unlikely. In such a situation, even if one strategy is far superior to another, its fraction of deadlines metric will not be very different because it will be dominated by failures that are due to a lack of resources. In other words, the predictor sensitivity will be low. Figure 6.5(a) shows that this is indeed that case. At very low slack levels such as 0 to 0.25, all but the RANDOM strategy perform similarly, while RANDOM is considerable inferior.

As the slack increases, scheduling feasibility and predictor sensitivity both increase resulting in both a larger fraction of deadlines being met and a greater differentiation between scheduling strategies. For the 4LS scenario, as slack increases from 0.25 to 1.25, we can see from Figure 6.5(a) that this is indeed
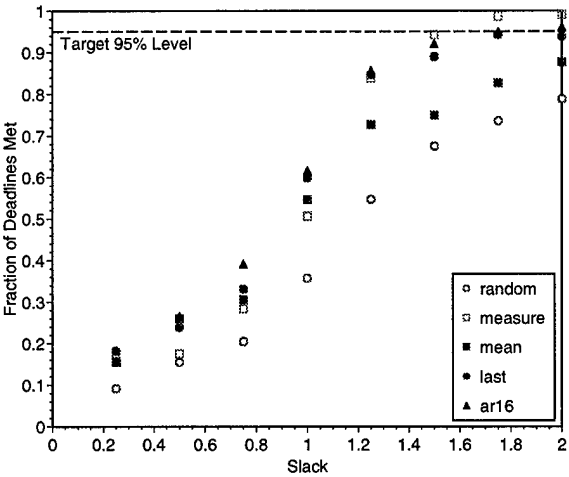
(a) fraction of deadlines met
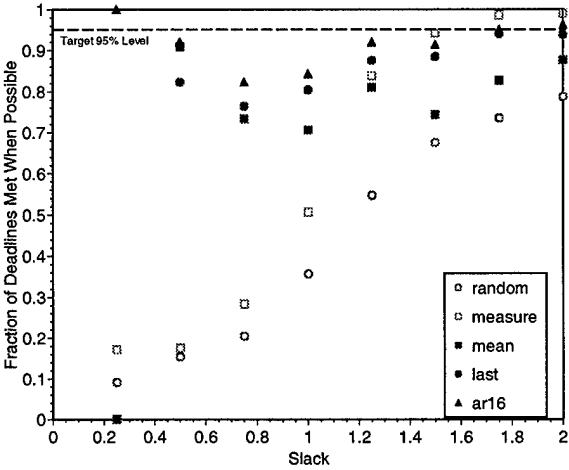
(b) fraction of deadlines met when possible
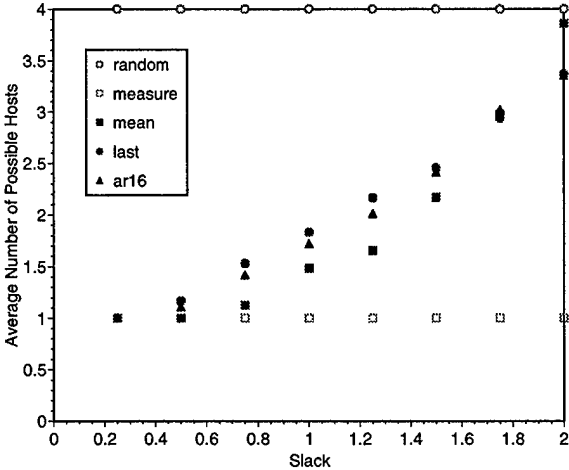
(c) number of possible hosts

Figure 6.5: Dependence of metrics on slack, 4LS scenario, 0 to 10 second tasks.

what is happening. For slacks from 0.25 to 1.0, the prediction-based strategies prove to meet 10-15% more deadlines than the MEASURE strategy and about 10-30% more than the simplistic RANDOM strategy. At slacks from 0.5 to 0.75, the AR(16) strategy results in 5-10% more deadlines being met than its closest competitor.

Beyond the critical slack, which is in the range 0.75–1 for the 4LS scenario, the scheduling feasibility continues to increase, but the predictor sensitivity begins to decline. Encountering a host on which a deadline can be met is becoming so likely that even a simple strategy is able to choose an appropriate host. For example, at slacks from 1.25 to 2.0 we can see from Figure 6.5(a) that the simple MEASURE predictor is performing as well as all the other strategies. The performance of MEASURE is always considerably better than that of RANDOM. As slack increases in that range, the AR(16) reaches its target level of 95% and the MEASURE strategy, which has no such level, surpasses it. It is important to note, however, that even at such high slack levels, the RANDOM strategy lags behind the other strategies by 20%. It is clear that even for such agreeable regimes, it is necessary to at least measure host load in order to achieve reasonable performance. The additional overhead to introduce prediction into a measurement based strategy is trivial and makes it much more resilient when slack declines.

The fraction of deadlines met is the metric that applications care about, and we have seen that prediction-based strategies, particularly AR(16), can improve that metric. However, it is the combination of the scheduling strategy, the requested slack, and the environment which determine the fraction of deadlines met. Our second metric, the fraction of deadlines met when possible, is determined solely by the strategy. For the prediction-based strategies, the advice of the real-time scheduling advisor includes a predicted running time for the task. If the advisor predicts that the deadline will be met and it is not, that is clearly the fault of the advisor and not the environment.

Figure 6.5(b) shows how the fraction of deadlines met when possible metric depends on the slack. Because the RANDOM and MEASURE strategies do not report a predicted running time to the application, their curves are identical to their fraction of deadlines met curves—we assume they always tell us the deadline could be met. Except for the fact that the greatest differentiation between the prediction-based strategies occurs at roughly the same slack as before, the trends here are remarkably different than before.

At low slacks, for the prediction-based strategies, the fraction of deadlines met when possible is high. Because the scheduling feasibility is low, it is rare to find a host on which the deadline can be met. However, the predictor sensitivity is low and so when such a host is encountered, it is easily discovered. Of course, the non-predictive MEASURE strategy will also likely discover it, but that strategy can not tag it specifically as being a situation where the deadline can be met because it has no idea what the execution time may be. From the point of view of the application, MEASURE and RANDOM always claim that the deadline can be met, while MEAN, LAST, and AR(16) make this claim only when they have good reason to believe that it is the case. Because they are able to report that majority of cases where the deadline cannot be met due to low slack, and prediction is easy (due to low predictor sensitivity) in that minority of cases where the deadline can be met, the fraction of deadlines that are met when they believe it is possible is quite high. This is what we see at low slack values in Figure 6.5(b).

As slack increases, scheduling feasibility and predictor sensitivity increases. This produces two effects. First, prediction error becomes an increasingly important cause of missing a deadline. Prediction error is independent of slack (recall that the running time advisor is unaware of slack), so this implies that the prediction strategies are increasingly prone to reporting that they can meet a deadline when they cannot. The second effect is that differences in prediction error lead to more differentiation between the different predictive strategies as slack increases. Figure 6.5(b) shows both of these effects. There is a dip in the performance of the predictive strategies as slack increases from 0 to 1, and their performance becomes increasingly different. Notice that the AR(16) strategy has the best overall performance by the fraction of deadlines met when possible metric. Even better prediction-based strategies would lead to even less of a dip

in performance around the critical slack.

Beyond the critical slack, further slack increases lead to decreasing predictor sensitivity, while the scheduling feasibility continues to increase. This results in a turn-around for the predictive strategies because failures to meet deadlines become less dependent on prediction error and thus they are less likely to report that a deadline can be met when it in fact can not. Furthermore, there is less differentiation between the different predictive strategies because the predictor sensitivity is declining. Figure 6.5(b) at slack levels from 1 to 2 illustrate these effects.

The prediction-based strategies can trade slack for randomness in their scheduling advice. Higher slacks mean that more hosts can be identified on which a task can meet its deadline. The prediction-based strategies then chose randomly from among the possible hosts, decreasing the change of collision with other advisors. In contrast, the RANDOM and MEASURE strategy represent extremes in the amount of randomness introduced. RANDOM introduces the maximum randomness, completely disregarding the task deadline, while MEASURE (in almost all cases) thinks there is only one possible host for the task. The prediction-based strategies represent a useful middle ground, then, introducing as much randomness as is possible given the slack, while still attempting to meet the individual task's goals. Furthermore, the prediction-based strategies, especially AR(16) are usually better at meeting the individual task's deadline than the randomness-squandering MEASURE.

Figure 6.5(c) shows how the degree of randomness in the advisor's advice varies with nominal time for the five strategies. The metric is the average number of possible hosts the strategy chooses from. At low slack levels, the randomness of the prediction-based strategies converges with that of MEASURE, while at high slack levels, it converges with RANDOM. Notice that even at the critical slack level around 1.0, where is is just becoming possible, in expectation, to meet the deadlines, the best prediction-based strategy chooses from approximately two hosts on average, introducing at least that degree of randomness.

Performance versus nominal time

The performance of the different scheduling strategies also depends on the nominal time of the task. For simplicity, the model of Section 6.5 assumed that the prediction error was independent of the nominal time, but as we showed in the previous chapter, this is not the case. To assess the effect of nominal time on the performance metrics, we will consider the 4LS scenario in detail, looking at both all recorded slacks and at those critical slacks where, according to the model and the results presented in Figure 6.5, the predictor sensitivity is highest.

Figure 6.6 shows the dependence of the performance metrics on the nominal time for the 4LS scenario. To produce the figure, testcases of all slacks from 0 to 2 were used. A point represents the interval of nominal times between it and the preceding point. For example, the fraction of deadlines met for nominal times from 0.1 to 1 second using the AR(16) strategy is plotted as a triangle at (1, 0.95) in Figure 6.6(a). Each point on the graph represents approximately 320 testcases.

The I/O boost provided by the OS scheduler benefits shorter tasks more than longer tasks, as we discovered in the previous chapter. The other load on a host, even if it is considerable as with the 4LS hosts, has little effect on a sub-second task, and thus most of these tasks meet their deadlines, even using very simple strategies. For longer tasks, the other load becomes increasingly visible, and the chance of a deadline being met declines. Consider the fraction of deadlines met by the RANDOM strategy as plotted in Figure 6.6(a). Sub-second tasks have a greater than 80% chance of meeting their deadlines. As the nominal time climbs, the chance of meeting the deadline declines until, for 4 to 10 second tasks, it bottoms out to about 40%, roughly half as good.

Even with the I/O boost, small tasks benefit from a more sophisticated strategy. With MEASURE, subsecond tasks have a 95% chance of meeting their deadlines, which is an improvement of 15% over RAN-
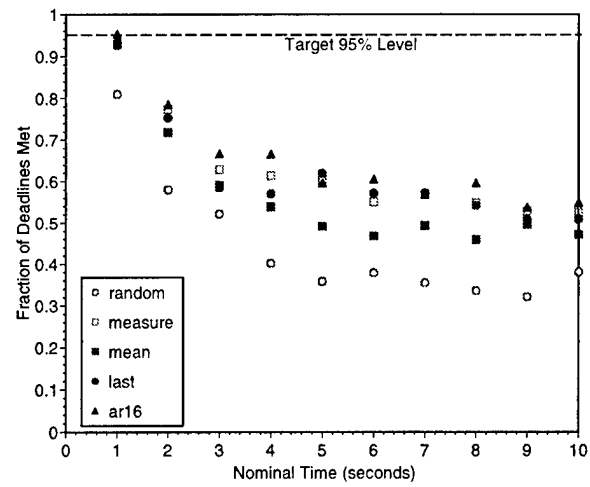
(a) fraction of deadlines met
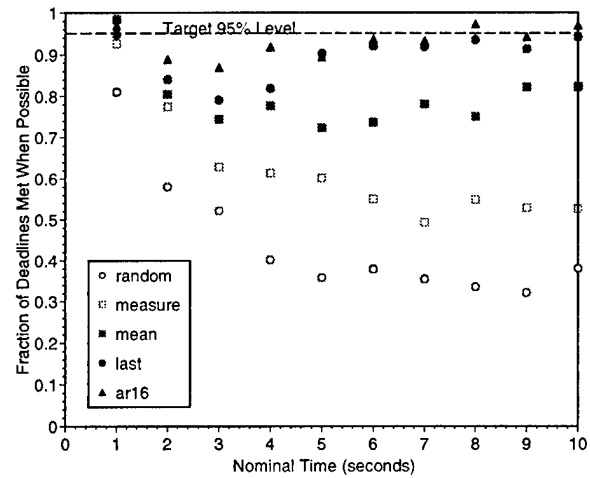
(b) fraction of deadlines met when possible
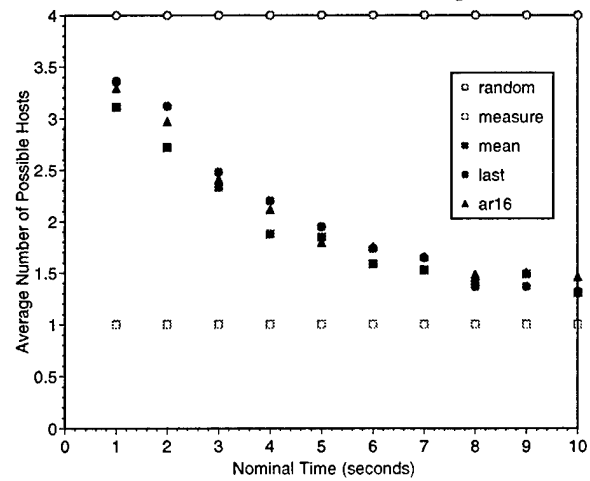
(c) number of possible hosts

Figure 6.6: Dependence of metrics on nominal time, 4LS scenario, 0 to 2 slack.

DOM. However, there is not much additional benefit, by this metric, to using the prediction-based strategies for these short tasks. As the nominal time increases, however, we can see that the prediction-based strategies, except for the MEAN predictor, perform better than MEASURE. As we can see from Figure 6.6(a), at nominal times from 3 to 4 seconds, the AR(16) strategy manages to make 68% of its tasks meet their deadlines, while the MEASURE strategy manages only 62%. Overall, the AR(16) strategy performs at least as well as the MEASURE strategy and usually has better performance than the LAST strategy. It is clearly desirable, for any nominal time, to use the MEASURE strategy instead of the RANDOM strategy. The prediction-based strategies, especially AR(16), make the chance that a deadline will be met even greater at a very small additional overhead, compared to the MEASURE strategy.

As we discussed earlier, the prediction-based strategies are able not only to suggest an appropriate host for a task, but are able also to assert whether a deadline can be met on that host or not. Because of this, we can determine how well these strategies perform when they actually believe that sufficient resources are available to meet deadlines, which we measure by the fraction of deadlines met when possible metric. This is a metric of considerable interest to applications because it determines to what extent they should trust the advisor when it asserts that they can meet the deadline by using a particular host. Figure 6.6(b) shows how the performance of the different strategies varies with the nominal time of the task on the 4LS scenario. Because the RANDOM and MEASURE strategies always implicitly assert that the deadline can be met, their curves are identical to those in Figure 6.6(a).

For the prediction-based strategies, the fraction of deadlines met when possible metric is much less dependent on the nominal time than the fraction of deadlines met metric (compare Figure 6.6(b) with (a)). As we noted earlier, it is also much less dependent on the slack. Clearly, the prediction-based strategies go a good job of determining not only an appropriate host for meeting a task's deadline, but also whether the deadline will actually be met on that host.

Although all of the prediction-based strategies consistently outperform RANDOM and MEASURE in terms of the fraction of deadlines met when possible metric, it is also clear that the LAST and AR(16)-based strategies perform much better than MEAN for all but the smallest tasks. Furthermore the AR(16) strategy is at least as good as LAST in all cases, and, for a considerable range of running times (1 to 4 second tasks), it considerably outperforms LAST. Recall that the case was similar with respect to different slack times. Clearly, AR(16) is the preferable strategy by this metric.

Figure 6.6(c) shows how the final metric, the average number of possible hosts, varies with the nominal time. Not surprisingly, because short tasks are essentially oblivious to other load on any host, and because the running time advisor is able to predict this obliviousness, the real-time scheduling advisor is able to introduce considerable randomness into its choice of hosts. For very small tasks, the choice is nearly as random as that of the RANDOM strategy. As tasks increase in size, there are fewer options as to where to schedule them, and so the advisor constrains the randomness it introduces. However, even for large 9 to 10 second tasks, the advisor can be considerably more random using a prediction-based strategy than using the MEASURE strategy. We noticed earlier that a similar pattern held true as the slack was reduced.

Clearly, the prediction-based strategies allow us to introduce randomness into scheduling decisions while still keeping the scheduled task's best interested at heart. Furthermore, despite introducing this globally beneficial randomness, the prediction-based strategies, particularly the AR(16) strategy, are better able than the MEASURE strategy to choose a host to meet the task's deadline. Finally, the application can trust that when the prediction-based strategies, particularly AR(16), assert that a deadline can be met, it actually will be met. This enables the application to negotiate with the real-time scheduling advisor, finding a combination of nominal time and slack where the deadline can be met with a high degree of certainty.
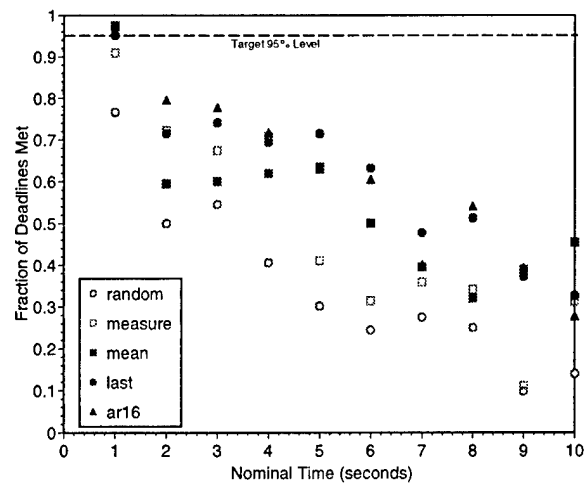
(a) fraction of deadlines met
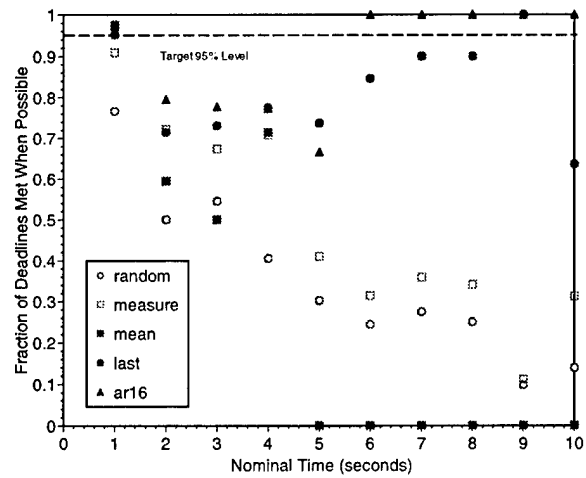
(b) fraction of deadlines met when possible
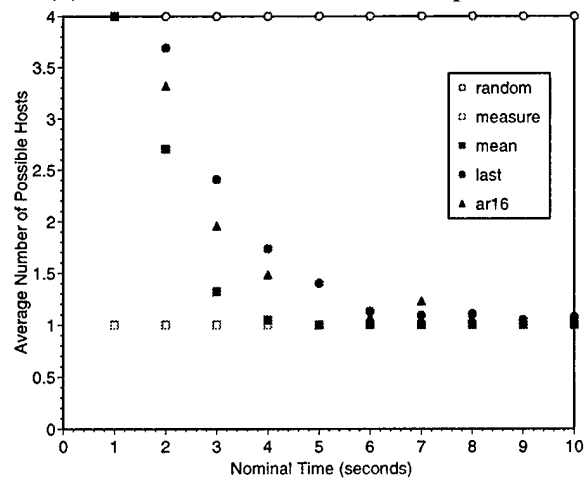
(c) number of possible hosts

Figure 6.7: Dependence of metrics on nominal time at critical slack, 4LS scenario, 0.75 to 1 slack.

Performance versus nominal time at critical slack

The critical slack is the interval of slack values where the predictor sensitivity is highest and the differentiation among predictors should be maximal. This is effectively where the slack is just large enough that deadlines can be met in expectation (ie, where one expects slightly more than 50% of deadlines to be met). From Figure 6.5, the critical slack appears to be 0.75 to 1.0 for the 4LS scenario. It is interesting to consider how the performance metrics vary with the nominal time for these slacks.

Figure 6.7 shows the dependence of the performance metrics on the nominal time for the 4LS scenario at the critical slack. To produce the figure, testcases of all slacks from 0.75 to 1 were used. A point represents the interval of nominal times between it and the preceding point. For example, the fraction of deadlines met for nominal times from 1 to 2 seconds using the RANDOM strategy is plotted as a circle at (2, 0.5) in Figure 6.7(a). Each point on the graph represents approximately 40 testcases. It is important to note that this is a sufficiently small number of testcases that some of the differences we speak about here are only statistically significant at lower confidence levels than our typical 95%. We will point out the cases where this is true. For all of the graphs AR(16) is better than RANDOM at a 95% confidence level.

At the critical slack, the prediction-based strategies are much better at meeting deadlines than the RANDOM or MEASURE strategies, and the AR(16) strategy performs best overall. As can be seen from Figure 6.7(a), the best prediction-based strategy results in 20 to 40% more deadlines being met than the RANDOM strategy, and from 5 to 30% more deadlines being met than the MEASURE strategy. As before, the fraction of deadlines met declines with increasing nominal time. The largest differentiation between the predictors occurs at medium sized tasks ranging from 3 to 7 seconds. With this small number of samples, the expected performance of AR(16) is only clearly better than MEASURE and the other prediction-based strategies with a lower confidence level of 75%.

Compared to its metrics over all slack levels (Figure 6.6(b)), the dependence of the fraction of deadlines met when possible metric on the nominal time is considerably different at the critical slack (Figure 6.7(b)). In particular, the dip in performance for medium-sized tasks is deeper and we see more extreme behavior on the part of various predictors. At the critical slack, the predictor sensitivity is highest, and so we would expect to see the greater differentiation among the prediction-based strategies, which we do. The overall dip in performance is due to the fact that, for very small tasks, the background load is not terribly important in meeting the deadline. For this reason, the error due to predicting the running time is also not important, and thus the prediction sensitivity is low.

As the nominal time increases the prediction sensitivity increases and we see more errors. Because we are at the critical slack, these errors are magnified because even small errors can lead the prediction-based strategies to erroneously conclude that the deadline can be met. As the nominal time continues to increase, all of the predictors begin to produce very large confidence intervals because the predictability of the load signal declines to its variance. With these large confidence intervals there are few cases where the prediction-based strategies can be met, and those occur when the load is so low that there is a very good chance that the deadline will be met. In the case of the MEAN strategy with nominal times greater than 4 seconds, there were no testcases where the strategy believed the deadline could be met, leading to the zero results. Statistically, AR(16) is better than MEASURE by this metric at a confidence level of 95% for nominal times of six seconds and better, and at a confidence level of 75% for times of two seconds and better.

Figure 6.7(c) shows how the average number of possible hosts depends on the nominal time at the critical slack. The graph is similar to the overall dependence presented in Figure 6.6(c) in that the amount of randomness introduced is very high for small tasks and declines as task size increases. Unlike Figure 6.6(c), however, there is a greater distinction between the different prediction-based strategies. LAST is the preferable strategy by this metric, although the differences between LAST and AR(16) have only a low statistical significance. Both AR(16) and LAST are statistically superior to MEASURE at a confidence level of 95%

for nominal times less than four seconds, however.

6.6.5 Other scenarios

In the previous section, we detailed how, for the 4LS scenario, the performance of the different strategies for the real-time scheduling advisor depended on the nominal time and the slack. In this section, we will report on similar studies for the 4SL, 4MM, 5SS, and 2CS scenarios (For direct comparison, we also include 4LS scenario, restricted to slacks from 0 to 1). The intent here is to illustrate how the performance metrics vary from scenario to scenario as well as with slack and nominal time. Because this introduces far too much information when presented in the format of the previous section, we divide the nominal time and slack into fewer intervals and use a simpler bar graph format in this section.

### Overall performance for each scenario

The data from which this section's graphs and discussion were collected using the 4LS, 4SL, 4MM, 5SS, and 2CS scenarios. For each scenario, we ran 8000 testcases whose nominal times were selected randomly from 0.1 to 10 seconds, and whose slacks were selected randomly from 0 to 1. It is interesting to consider the performance metrics of these testcases irrespective of nominal time and slack to ground our understanding of how the scenarios differ. Figure 6.8 presents the overall performance metrics for each scenario. Each bar represents the result of approximately 1600 testcases.

In terms of the fraction of deadlines met metric, plotted in Figure 6.8(a), the most obvious difference between the scenarios is that the performance of all the strategies is considerably lower for the 4LS scenario than for the other four scenarios. Recall that the 4LS scenario includes four hosts that all have a mean load around 1.0 and, as we discussed earlier, the critical slack is in 0.75 to 1.0. Because the upper bound of the slack values considered here is 1.0, most of the 4LS testcases can be expected to fail. In contrast, the hosts in the other scenarios either all have much lower load (4SL, 5SS, 2CS) or include a mixture of machines with high and low load (4MM), and so a much larger fraction of their testcases can be expected to have their deadlines met. As we might expect, the 4MM scenario is more difficult than the 4SL, 5SS, and 2CS scenarios because it includes two hosts with higher load.

It is also significant to consider how the fraction of deadlines met metric varies across the scenarios for the RANDOM strategy. For the 4LS and the 4MM scenarios, RANDOM achieves only slightly more than half of the performance of the best strategy. In the 4LS scenario, as we argued in the previous section, this is because a considerable fraction of the testcases occur at slack values that are near the critical slack where predictor sensitivity is high. In the 4MM case, the explanation is more prosaic: in expectation, two of the hosts (the ones with high mean load) will be "bad", while two (the ones with low mean load) will be "good". RANDOM will then pick a "bad" host half of the time, often resulting in a failure to meet the deadline.

The ability to distinguish between hosts that differ in such a simple way is why the MEASURE strategy is able almost always to outperform the RANDOM strategy on the fraction of deadlines met metric. From Figure 6.8(a), we can see that the prediction-based strategies are generally able to improve on this performance by a small to moderate amount. In particular, the AR(16) strategy is consistently at least as good as the MEASURE and RANDOM approaches. We will find later that, just as with the study of the 4LS testcase, the differences between the different strategies in terms of the fraction of deadlines met metric depends on the slack and nominal time. Near the critical slack and for sufficiently large nominal times, the difference between the prediction-based strategies, particularly AR(16), and MEASURE or RANDOM can be quite significant.

Figure 6.8(b) shows the fraction of deadlines met when possible metric for each of the scenarios under each of the strategies. Using this metric, which reflects how much the application can trust the advisor when it claims that a deadline can be met on a particular host, the prediction-based strategies are clearly superior
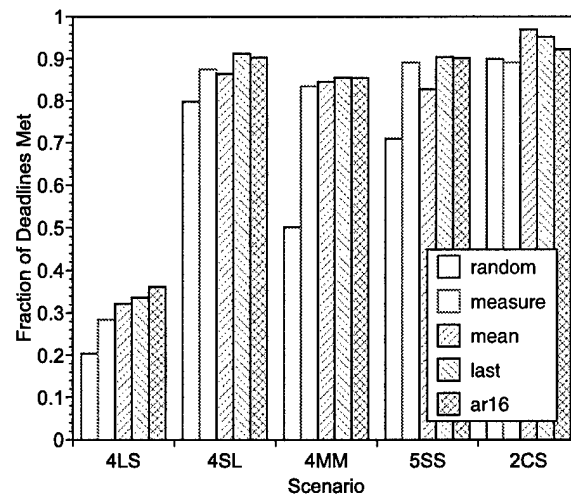
(a) fraction of deadlines met
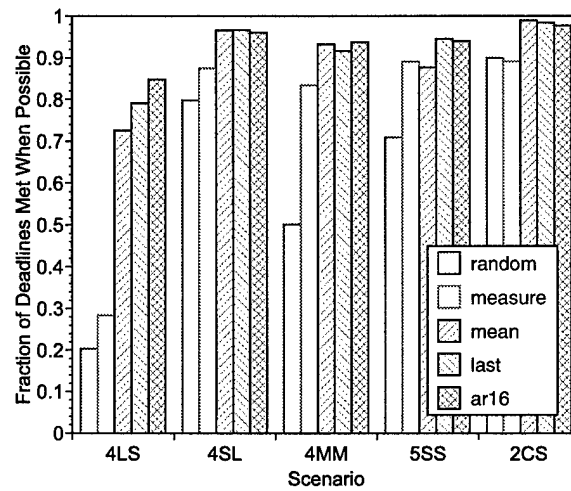
(b) fraction of deadlines met when possible
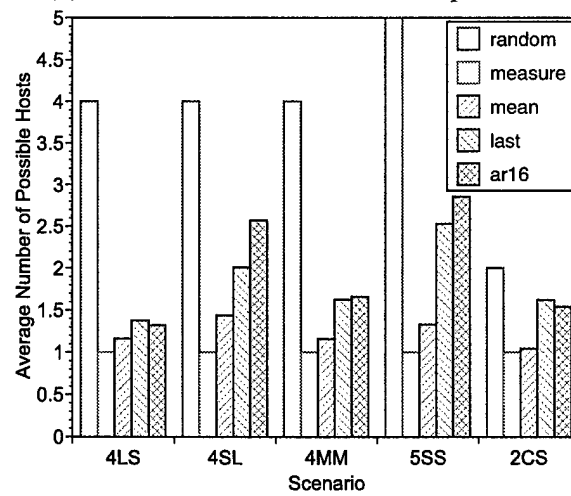
(c) number of possible hosts

Figure 6.8: Overall scheduling results, 0 to 10 second tasks, 0 to 1 slack.

to the RANDOM and MEASURE strategies. Furthermore, for the more difficult scenario, 4LS, the AR(16) strategy is clearly superior to the other prediction-based strategies, while it is at least their equal for the other scenarios. We will also find that these differences are accentuated with tighter slack times and longer nominal times.

Figure 6.8(c) shows the number of possible hosts metric for each of the scenarios and strategies. As we can see, the prediction-based strategies always allow the advisor to introduce more randomness into its decisions than the MEASURE strategy. Furthermore, the AR(16) strategy is able to introduce the most randomness in most cases. Remarkably, this extra randomness, which reduces the chance that competing advisors will contend for the same host, comes at no cost (and often a benefit) in terms of the fraction of deadlines that are met and almost always a benefit in terms of the fraction of deadlines met when possible, the application metrics. In other words, using the prediction-based strategies, the real-time advisor can avoid contending with other real-time advisors while simultaneously providing a real benefit to the application.

Performance versus slack

As we discovered in our in-depth study of the 4LS scenario, the performance of the real-time scheduling advisor depends strongly on the slack time. As slack nears a critical level, the performance of the prediction-based strategies becomes significantly better than the MEASURE strategy. Furthermore, we begin to see greater differences between the predictors because the prediction sensitivity is high. In this section, we show how the performance metrics vary with slack for all of the scenarios. To avoid an overload of information, we'll use the format of Figure 6.9 to present each metric. Graph (a) will contain testcases for slacks in the range 0.0 to 0.33, graph (b) will cover slacks in the range 0.33 to 0.67, and graph (c) will cover 0.67 to 1.0. Thus, by looking down the column of graphs, the effect of greater slack times can be seen. A bar represents approximately 534 testcases.

As we can see from Figure 6.9, the fraction of deadlines met metric for all of the scenarios generally has the same relationship we discovered in our detailed analysis of the 4LS scenario. The difference is the location of the critical slack. For all of the other scenarios, the critical slack is clearly somewhere in the range 0.0 to 0.33. If we look at these scenarios' testcases at or below this critical slack (Figure 6.9(a)), we see that the prediction-based strategies generally outperform the MEASURE strategy, and that the AR(16) strategy is often preferable. However, although the case for the prediction-based strategies is strong for all the strategies, the AR(16) strategy is not clearly preferable. In fact, for the 2CS scenario, although it is at least as good as RANDOM, and considerably better than MEASURE, it does not work as well as the long term MEAN strategy. This suggests that in some scenarios, it may be worthwhile to run multiple strategies and use a multiple expert style algorithm to decide which the is preferable strategy at any one time. The multiple expert approach is used in the Network Weather Service prediction system [140], and has theoretical justification (cf. [21]). It is interesting to note that in the 2CS scenario, MEASURE is actually inferior to RANDOM. The behavior of these two hosts must be such that short-term behavior is very deceptive of even short term prospects.

At slacks ranging from 0.33 to 0.67 (Figure 6.9(b)), all of the scenarios other than 4LS have passed their critical slack level and there is little difference between MEASURE and the prediction-based strategies. However, the RANDOM strategy continues to perform badly, especially for cases which have hosts of widely differing mean load (4MM and 5SS). In this situation, the prediction-based strategies would provide the same performance as measure, but would be able to correctly inform applications whether a deadline could be met. Furthermore, the prediction-based approaches would be able to introduce more randomness into their scheduling decisions. This continues to hold true for slacks ranging from 0.67 to 1.0 (Figure 6.9(c)). Even with considerable slack, the RANDOM strategy is still confused by the very different hosts in the 4MM scenario. Also, we have now reached the critical slack for the 4LS scenario and we can
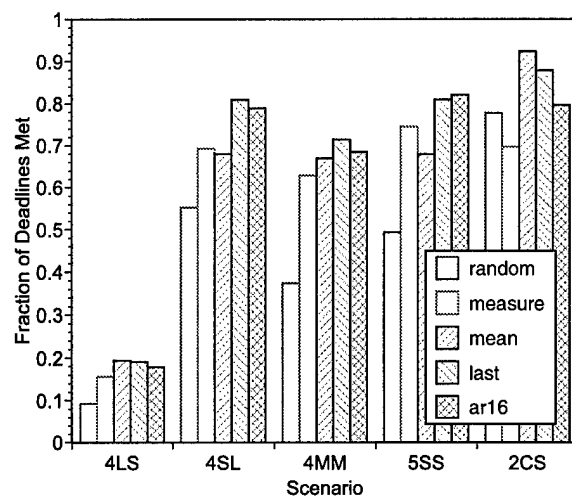
(a) slack 0.0 to 0.33
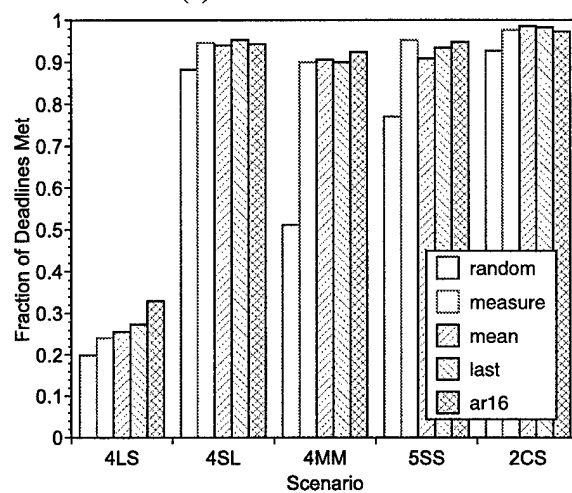
(b) slack 0.33 to 0.67
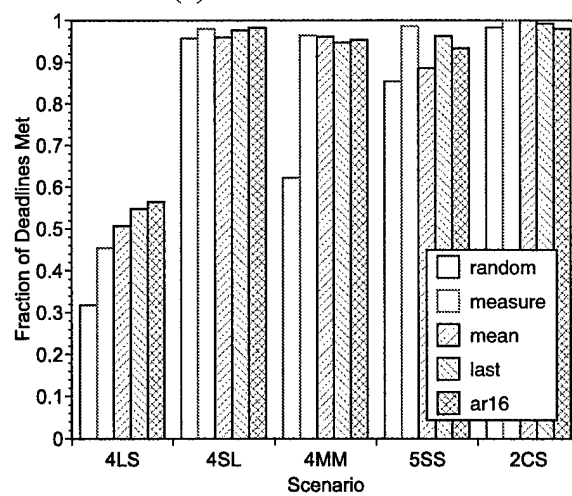
(c) slack 0.67 to 1.00

Figure 6.9: Fraction of deadlines met versus slack, 0 to 10 second tasks.

see the great differentiation between the different strategies which we noted before.

The fraction of deadlines met when possible metric also depends on the slack. Figure 6.10 shows this dependence for the different values of slack. At low slacks of 0.0 to 0.33 (Figure 6.10(a)), we can see that the predictive strategies are able to almost always correctly call whether their task to host assignment will in fact meet its deadline. Furthermore, the AR(16) strategy is usually able to do this best job of this. Recall that the target level of the predictive strategies here is 95%. The AR(16) strategy is able to achieve that desired level of accuracy on all the scenarios. The occasional missing bars for the MEAN scenario are due to the MEAN scenario never encountering a testcase which it believed could have its deadline be met. Clearly, at these low slack levels (recall that a 0.33 slack represents a deadline of 133% of the nominal time of the task), the predictive strategies, and AR(16) in particular, can give an application strong piece of mind when it asserts that a deadline can be met.

Figure 6.10(b) and (c) show how the fraction of deadlines met when possible metric fares for 0.33 to 0.67 and 0.67 to 1.0 slacks, respectively. In the case of 4LS, we are just on the edge of the critical slack, so the predictive strategies still offer a significant gain over MEASURE and AR(16). For the other scenarios, we see that MEASURE, and, for some, RANDOM, catch up with the prediction-based scenarios with sufficient additional slack. The advantage of the prediction-based strategies, and, in particular, AR(16), is that they deliver a (high) fraction of deadlines met when possible metric relatively independently of slack.

Figure 6.11 shows how the number of available hosts metric varies with the slack. As we showed in our analysis of the 4LS scenario, the prediction-based strategies use excess slack to introduce contention-avoiding randomness into the scheduling advice they provide with no detriment to the application. In fact, the application usually benefits, and even where the application benefits the most (near the critical slack), considerable randomness can be introduced. Figure 6.11(a) shows the number of possible hosts for 0.0 to 0.33 slacks. At this level, only a tiny amount of extra randomness is introduced, and that mostly by the AR(16) strategy. At slacks ranging from 0.33 to 0.67 (Figure 6.11(b), considerably more randomness can be introduced into each of the scenarios, and AR(16) usually introduces the most randomness. At high slack values ranging from 0.67 to 1.0, AR(16) is often able to introduce almost as much randomness as the RANDOM strategy while meeting more deadlines (eg, 5SS and 2CS scenarios). Notice that for the 4MM scenario, AR(16) chooses from roughly two hosts on average. These are most often the two lightly loaded hosts. If the slack were increased to 2.0, it would choose from all four of them.

Performance versus nominal time

The performance of the different strategies on the different scenarios also varies with the nominal time of the task to be scheduled. In general, the fraction of deadlines met declines with increasing nominal time, while the fraction of deadlines met when possible metric remains relatively stable, although it increases slightly with nominal time. Because smaller tasks "see" less of the background load on hosts due the disproportionate effect of the I/O boost on them, it is possible to introduce more randomness, measured by the average number of possible hosts, as nominal time decreases.

Figure 6.12 shows how the fraction of deadlines metric as the nominal time varies from (a) 0.1 to 3 seconds, (b) 3 to 6 seconds, and (c) 6 to 10 seconds. The graphs include slacks from 0 to 1, and each bar represents approximately 534 testcases. The decline in deadlines met with increasing nominal time is most marked for the 4LS scenarios because most of the testcases here are gathered for slacks below the critical slack. For the other scenarios, most of the testcases are above the critical slack, and so the decline is not as clear, although it is certainly noticeable. It is always easier to schedule short tasks to meet their deadlines. We can see that in all cases, there is a prediction-based strategy that performs at least as well as the MEASURE strategy, and that strategy is usually the AR(16) strategy. For the 4LS and 4SL cases, the performance gain of the AR(16) strategy over the MEASURE strategy is most clear. In the next section,
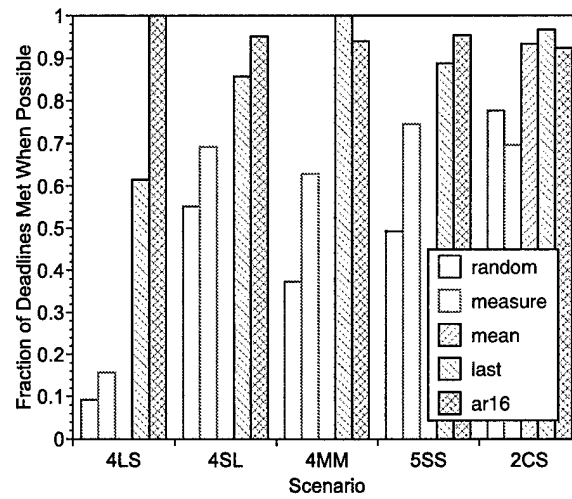
(a) slack 0.0 to 0.33
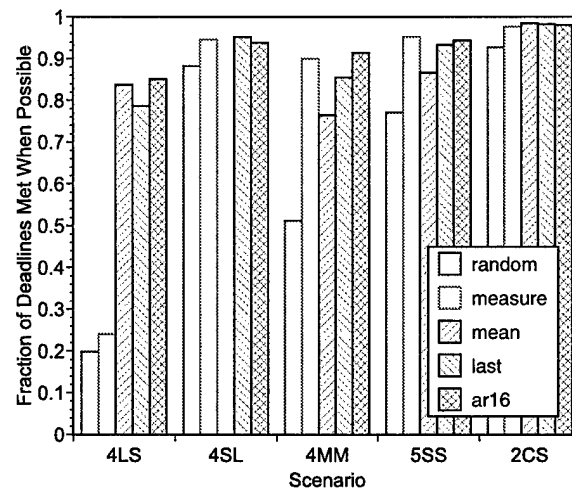
(b) slack 0.33 to 0.67
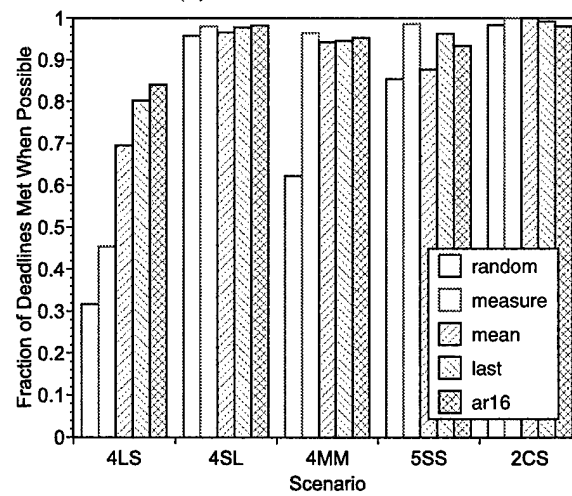
(c) slack 0.67 to 1.00

Figure 6.10: Fraction of deadlines met when possible versus slack, 0 to 10 second tasks.

(a) slack 0.0 to 0.33

(b) slack 0.33 to 0.67

(c) slack 0.67 to 1.00

Figure 6.11: Number possible versus slack, 0 to 10 second tasks.

(a) 0.1 to 3 second tasks

(b) 3 to 6 second tasks

(c) 6 to 10 second tasks

Figure 6.12: Fraction of deadlines met versus nominal time, 0 to 1 slack.

we will vary both the slack and the nominal time. This will show that there are more marked differences between different nominal times when slack is constrained. This is in tune with our in-depth analysis of 4LS in the previous section.

In terms of the fraction of deadlines met when possible metric, which represents the confidence that an application can have that a task will actually meet its deadline when the advisor says it can, the dependence on nominal time is much clearer. As Figure 6.13 shows, as the nominal time of the task increases, the application can be much more confident that the prediction-based strategies will be truthful than if it used the MEASURE or RANDOM strategies. Even for the 2CS case, where measurement performs worse than prediction in terms of meeting deadlines and the general ranking of predictors is opposite to that seen in the other scenarios, the prediction-based strategies are highly accurate when they claim a deadline can be met. As Figure 6.13(c) shows, for 6 to 10 second tasks, the prediction-based strategies are correct almost 100% of the time, while the next best strategy is accurate only 85% of the time. As we shall see in the next section, these distinctions are accentuated when the slack is constrained to be nearer the critical slack. Except for in the extremely resource-constrained 4LS scenario, the prediction-based strategies maintain a high and stable accuracy for all the nominal times.

Because short tasks benefit more from I/O boosts, they are easier to schedule. The prediction-based strategies are able to use this attribute to increase the amount of randomness they introduce into their scheduling decisions, all with little effect on the benefit that applications derive from the decisions. Figure 6.14 shows how our measure of that randomness, the average number of possible hosts, increases as the nominal time decreases. In almost all cases, the predictive strategies are able to introduce more randomness than the MEASURE strategy, and, for small nominal times, the amount they introduce becomes a significant fraction of that which the pure RANDOM strategy can introduce. It is usually the case that the AR(16) strategy is able to introduce the most additional randomness, although that is not always the case, especially for the 4LS scenario with short nominal times and for the 2CS case for long nominal times.

Performance versus slack and nominal time

The performance of the different strategies varies jointly by scenario, slack level, and nominal time. Previously, we looked at two dimensional projections of this dependence, looking at scenario and slack level, and then scenario and nominal time. Now we will look at the joint relationships, as we measured for our five scenarios. We shall use precisely the same bar graph technique as previously to graph the results. However, each bar graph will have both its slack and its nominal time constrained. To avoid generating too many graphs (with each bar representing too few testcases), we will look at only six combinations of slack level and nominal time. The slacks we will look at are 0.0 to 0.5 and 0.5 to 1.0. The nominal times will be the same as before: 0.1 to 3, 3 to 6, and 6 to 10 seconds.

Figure 6.15 shows how the fraction of deadlines met metric varies with nominal time and slack. From top to bottom, the graphs' time constraints increase, while from left to right, the graphs' slack constraints increase. First, let's consider the right column of graphs, where the slack levels range from 0.5 to 1.0. The 4LS scenario is clearly the most interesting here, because for the other scenarios slack levels above 0.5 make the scheduling feasibility sufficiently high and the predictor sensitivity sufficiently low that most tasks meet their deadlines with even a simple strategy. However, it is important to point out that even with these high slacks, the RANDOM strategy is often simply not sufficient to result in a large number of tasks meeting their deadlines. This is especially true for larger tasks—consider Figure 6.15(f), where three of the five scenarios show abysmal performance with the RANDOM strategy. Except for the 4LS scenario, the MEASURE strategy is generally sufficient for these higher slack situations, and the prediction-based strategies do not perform significantly better. For the 4LS scenario, it is also interesting to note that even though the slack is only near the critical slack, 80% of the deadlines for short tasks can be met.

(a) 0.1 to 3 second tasks

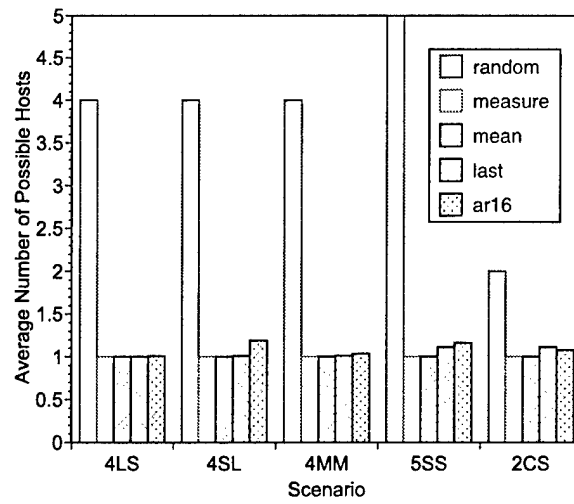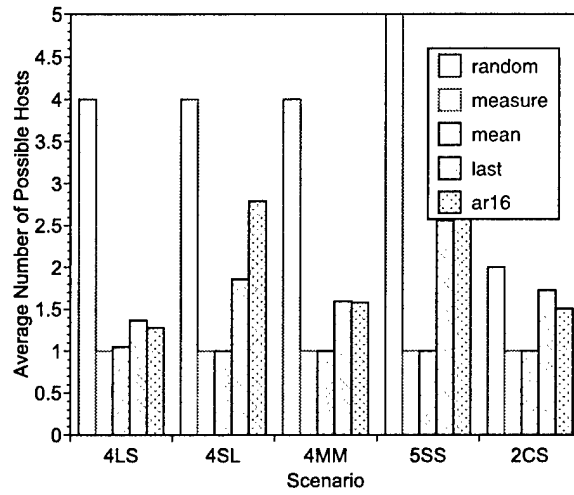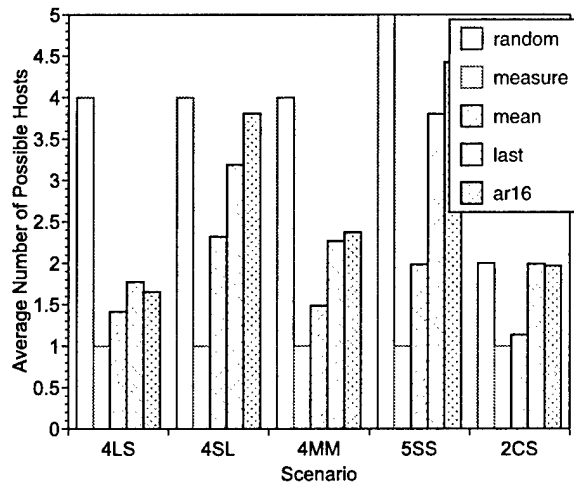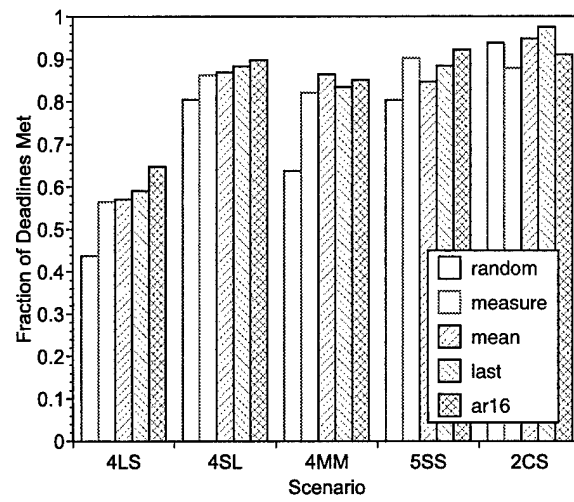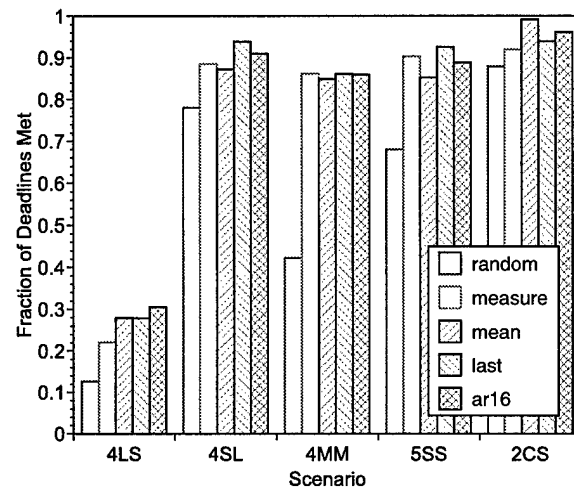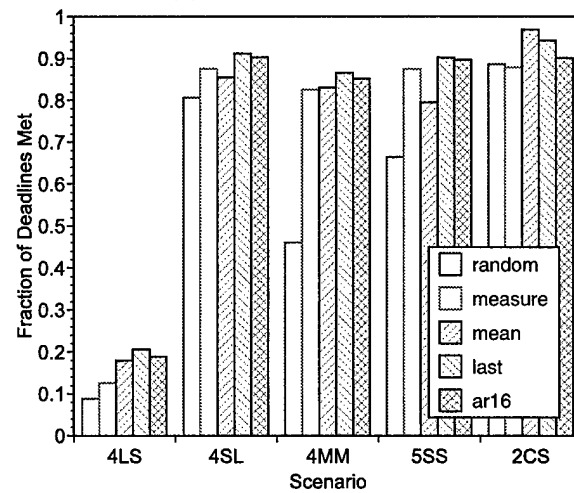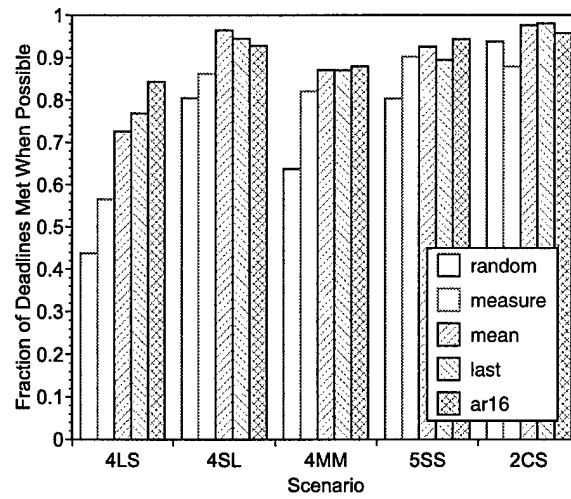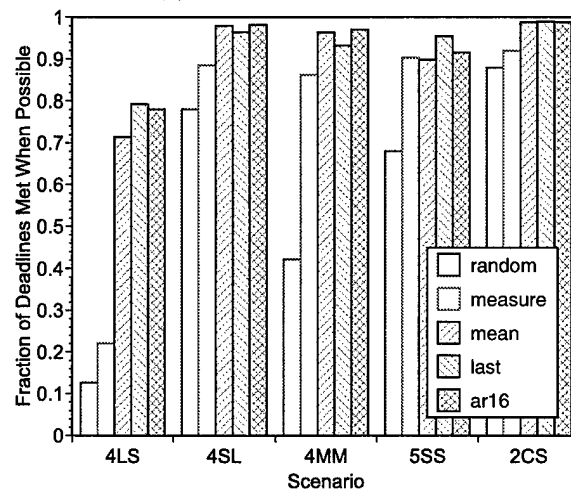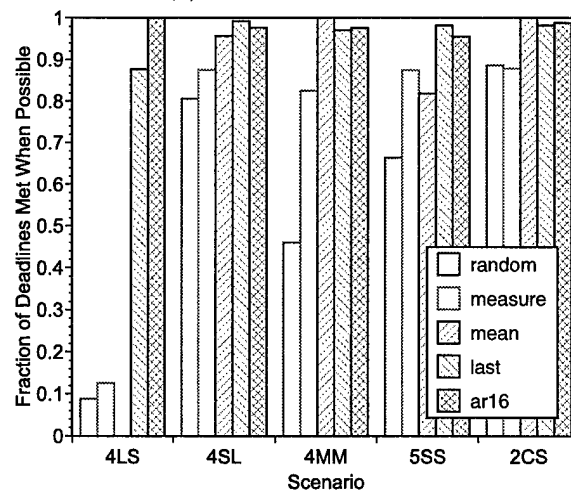(b) 3 to 6 second tasks

(c) 6 to 10 second tasks

Figure 6.13: Fraction of deadlines met when possible versus nominal time, 0 to 1 slack.

(a) 0.1 to 3 second tasks

(b) 3 to 6 second tasks

(c) 6 to 10 second tasks

Figure 6.14: Number of possible hosts versus nominal time, 0 to 1 slack.

(a) 0.1 to 3 second tasks, 0 to 0.5 slack

(b) 0.1 to 3 second tasks, 0.5 to 1.0 slack

(c) 3 to 6 second tasks, 0 to 0.5 slack

(d) 3 to 6 second tasks, 0.5 to 1.0 slack

(e) 6 to 10 second tasks, 0 to 0.5 slack

(f) 6 to 10 second tasks, 0.5 to 1.0 slack
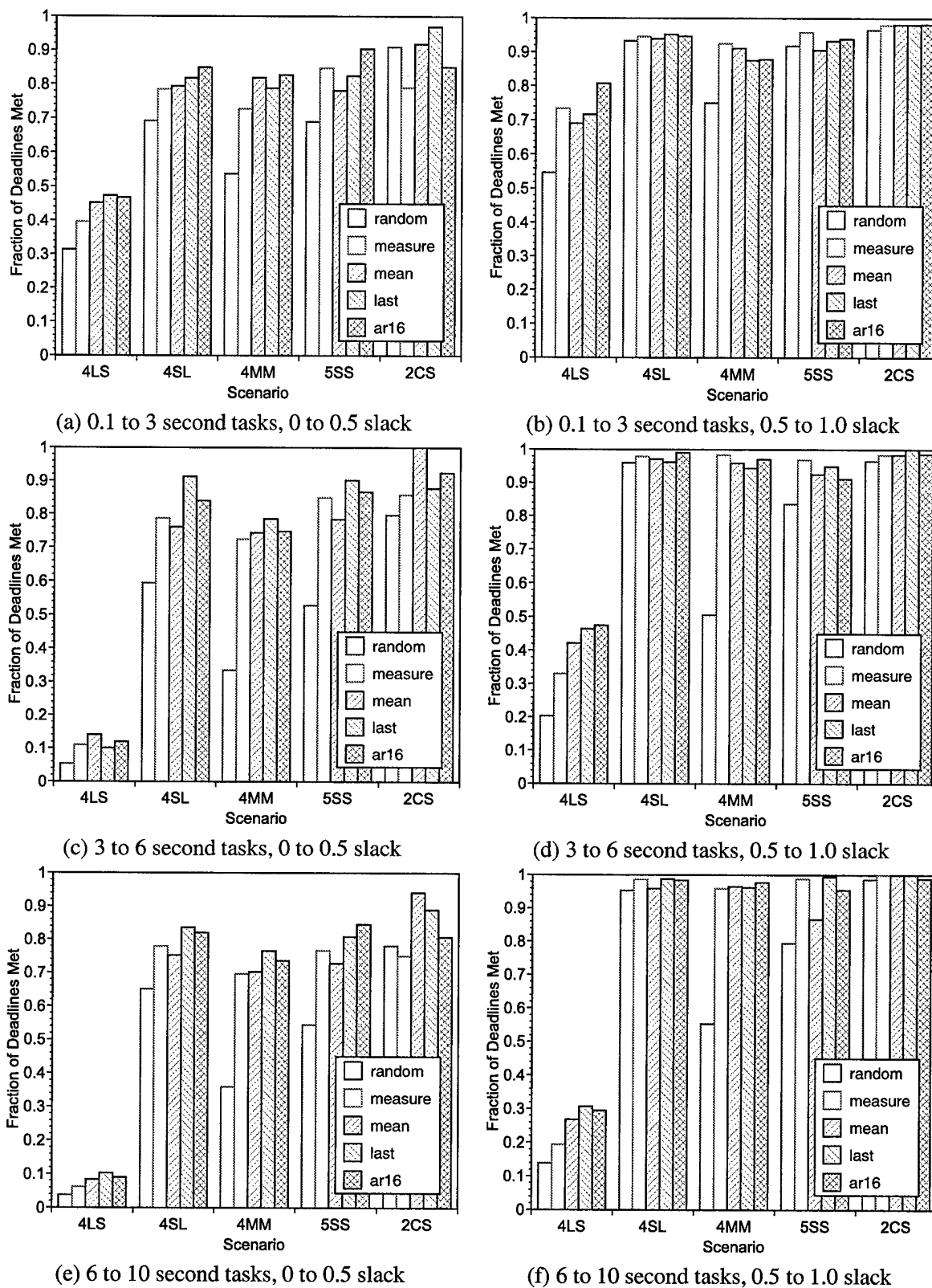
Figure 6.15: Fraction of deadlines met versus nominal time and slack, several scenarios.

It is when we consider tighter slacks that the benefit of the prediction-based strategies becomes clearer. The left column of Figure 6.15 shows what happens when the slack is constrained to 0.0 to 0.5. Now the prediction-based strategies generally out-perform the MEASURE strategy. In the 2CS case the RANDOM strategy proves to be better than the MEASURE strategy, and even here there always a prediction-based strategy that out-performs it. We can also see that the fraction of deadlines met declines as the nominal time increases, just as we showed earlier. The prediction-based approaches clearly provide better performance with low slacks for all nominal times.

Figure 6.16 shows the dependence of the fraction of deadlines met when possible metric on the nominal time and the slack. The graphs in the figure are arranged identically to those in Figure 6.15—all that is changed is the choice of metric. The main observation to make is that the prediction-based strategies, in particular the AR(16) strategy, are able to predict, with accuracy that remains consistently high irrespective of the nominal time and the slack level, whether the deadline will be met if the application follows their advice.

There is very little difference between the strategies at high slacks and short nominal times, other than that for three of the scenarios RANDOM proves to have inadequate performance. As the slack decreases, however, the prediction-based strategies maintain their accuracy. Oddly, the difference between the different predictors and between the predictors as a group and the MEASURE strategy appears to decline with increasing nominal time. This is not an effect that we saw by looking purely at the nominal time, independent of slack. Nonetheless, it is clear that the predictors are most beneficial, in terms of the fraction of deadlines met when possible, for low slack situations with short nominal times.

We would expect that the degree of randomness that the prediction-based strategies can introduce, as measured by the average number of possible hosts, grows as slack increases and nominal time decreases. Figure 6.17 shows that this is indeed what happens. The figure also shows that the AR(16) strategy is generally able to introduce the most additional randomness. Indeed, for 0.1 to 3 second tasks at 0.5 to 1 slacks (Figure 6.17(b)), it is able to introduce almost as much randomness as the pure RANDOM strategy in three of the five scenarios. Again, we point out that this extra randomness comes at no cost in terms of the application metrics, on which the prediction-based strategies consistently outperform the pure RANDOM strategy.

6.6.6   Contention between advisors

An important concern facing measurement- or prediction-based real-time scheduling advisors is that of contention between different advisors. In the design of the resource prediction-based real-time advisor system, as described in Chapter 1, all advisors observe the same predictions of signals that characterize resource availability (here the signal is host load, which measures CPU availability), but they do not coordinate their scheduling decisions. In the case of advisors using the MEASURE strategy, each advisor observes the same host load measurement. This lack of coordination helps to make measurement- or prediction-based real-time advisors scalable. However, it is conceivable that the advisors might all decide that the same host is most appropriate for their tasks. Those tasks would then contend for that host, possibly resulting in them all missing their deadlines. Of course, if this should happen, the load on the host will explode and, presumably, all the advisors would use different hosts for their next tasks. However, the advisors could all decide to use the same host *again*. In the case of something like the MEASURE strategy, one could imagine the advisors becoming synchronized, all choosing the host with the minimum load, beating it to death, then, for their next tasks, all choosing the same host, the current host with the minimum load.

Two factors ameliorate the chance of this sort of disastrous advisor synchronization occurring. First, for both kinds of strategies, scheduling requests are not likely to become synchronized or even correlated. Recall that a particular interactive application's tasks are generated in response to its user's actions, which

(a) 0.1 to 3 second tasks, 0 to 0.5 slack

(b) 0.1 to 3 second tasks, 0.5 to 1.0 slack

(c) 3 to 6 second tasks, 0 to 0.5 slack

(d) 3 to 6 second tasks, 0.5 to 1.0 slack

(e) 6 to 10 second tasks, 0 to 0.5 slack
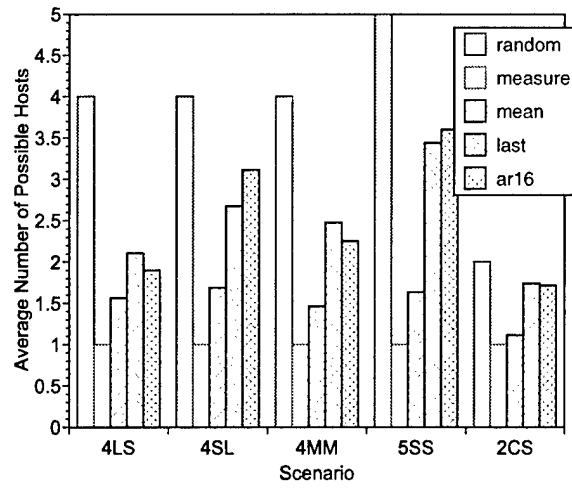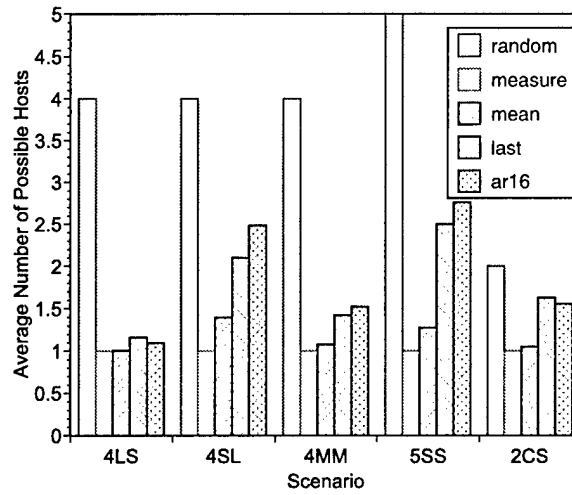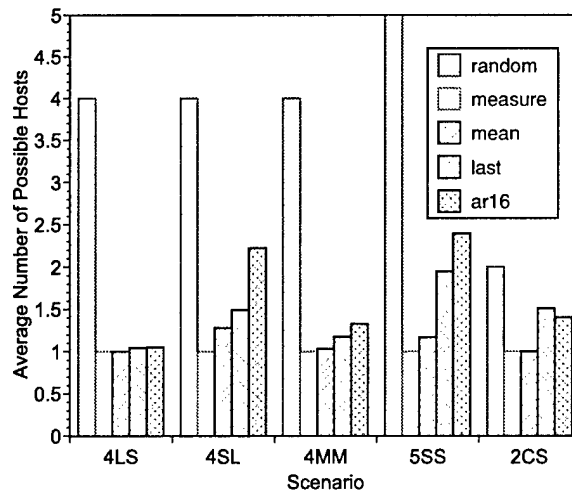
(f) 6 to 10 second tasks, 0.5 to 1.0 slack
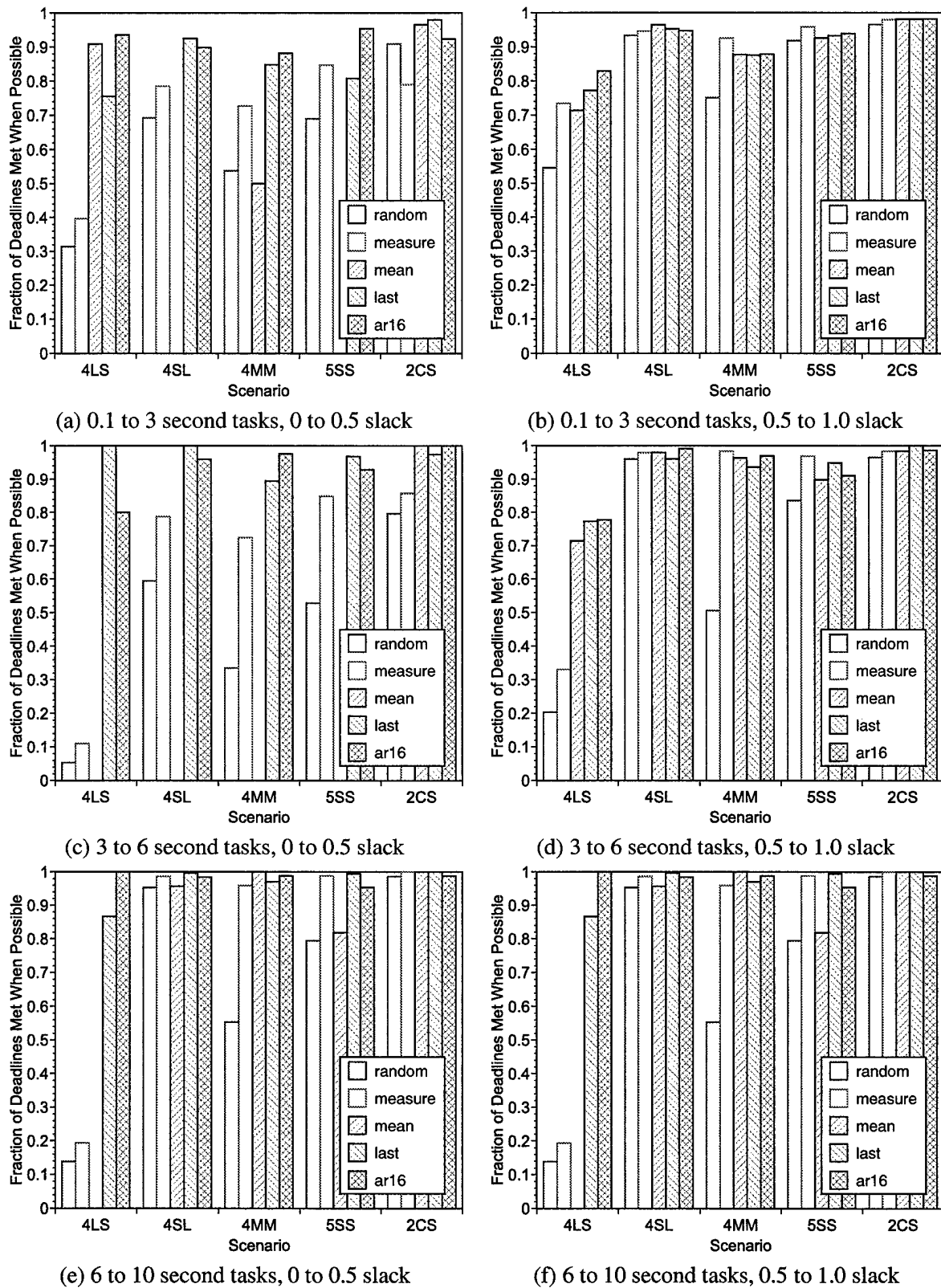
Figure 6.16: Fraction of deadlines met when possible versus nominal time and slack.

(a) 0.1 to 3 second tasks, 0 to 0.5 slack

(b) 0.1 to 3 second tasks, 0.5 to 1.0 slack

(c) 3 to 6 second tasks, 0 to 0.5 slack

(d) 3 to 6 second tasks, 0.5 to 1.0 slack

(e) 6 to 10 second tasks, 0 to 0.5 slack

(f) 6 to 10 second tasks, 0.5 to 1.0 slack

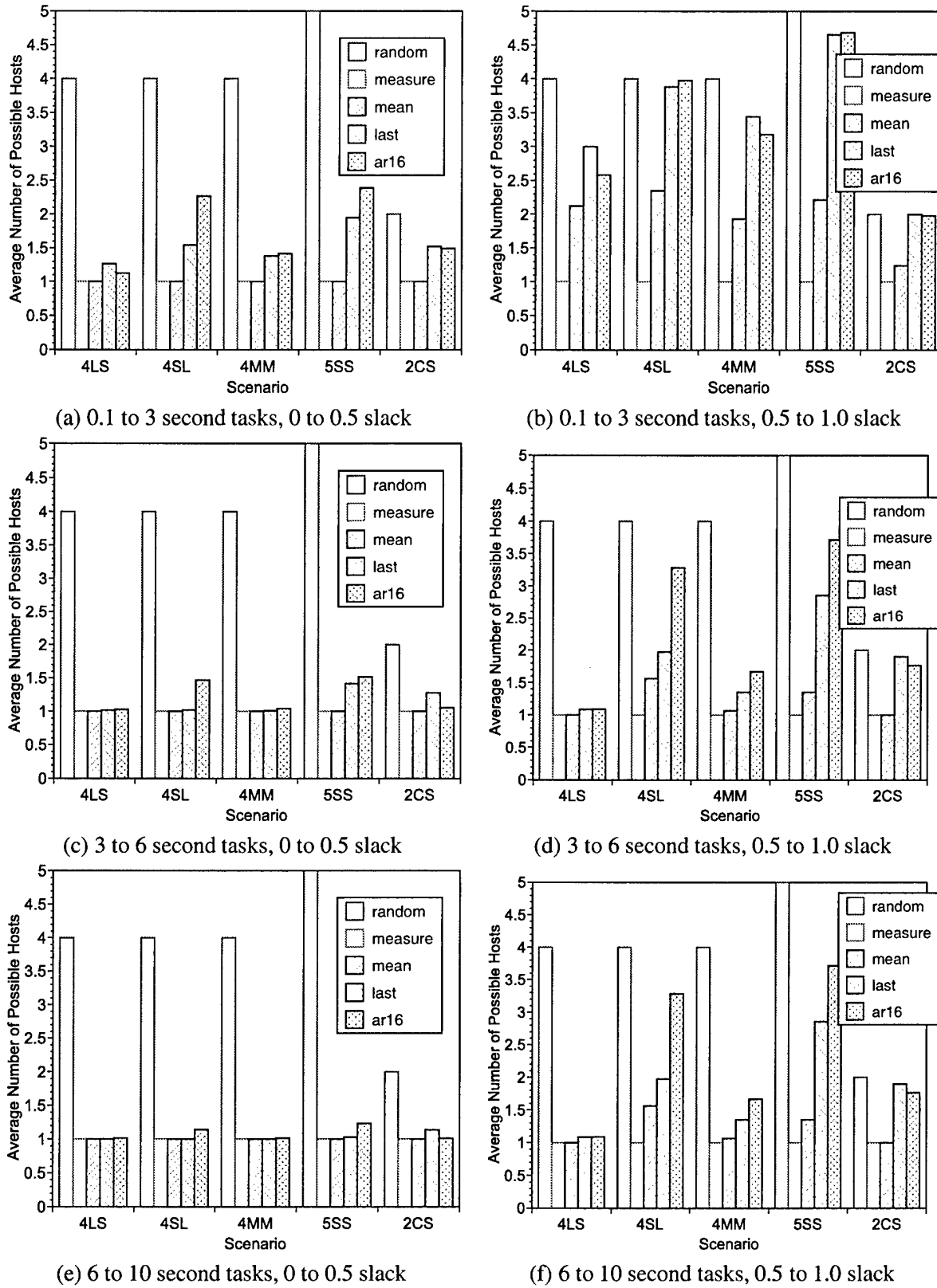Figure 6.17: Number of possible hosts versus nominal time and slack, several scenarios.

arrive asynchronously *after* the task initiated by the previous user action is completed. Thus, an interactive application has only a single task outstanding at any one time, and those tasks are not initiated synchronously with any clock, although their initiations may be correlated. If multiple interactive applications using real-time scheduling advisors are running, each is responding to a different user, and we would expect that the users' actions are not synchronized. These two sources of asynchrony should help to avoid a single collision of real-time scheduling advisors from synchronizing the advisors and resulting in a cascading chain reaction.

The second factor that limits the chance of disastrous advisor synchronization is limited to the prediction-based strategies. Unlike the MEASURE strategy, the prediction-based strategies can convert excess slack into randomness as to which host a task is sent to. This source of randomness can serve to break any synchronization that may be starting. Suppose, for example, that two advisors are working in a scenario that includes two hosts, where one host is always appropriate, while the second host is appropriate 20% of the time. This results in an average (expected) number of available hosts being 1.2. If the advisor chooses randomly between the two hosts when they are both appropriate, then it will pick the second host 10% of the time. Now suppose that the two advisors assign tasks at identical times. The chance that the two tasks will be both be assigned to the first host is $(.9)(.9) = 0.81$. The probability that two consecutive sets of tasks will have both tasks of each set assigned to the first host is $(.81)^2 = 0.65$. For n sets of tasks, the probability is $(.81)^n$—it declines exponentially with the length of the sequence. If the two hosts have roughly the same load, then the probability that each will be included in the set of appropriate hosts when a scheduling decision is made is 0.6 for the same expected number of available hosts $(1.2 = (0)(0.4)(0.4) + (1)(0.4)(0.6) + (1)(0.6)(0.4) + (2)(0.6)(0.6))$. In this case the probability of assigning both tasks of a set to the same host is $(0.6)(0.6) = 0.36$ and the probability of assigning n consecutive tasks to the same host is $(0.36)^n$.

To see if these two factors prevented repeated collisions in practice, we ran testcases using four competing scheduling advisors on the two host 2MP scenario. Each client ran 8000 randomized testcases as per the recipe in Section 6.6.2. The hosts have roughly the same load. Figure 6.18 shows the overall performance metrics for each client's testcases. The slacks included are from 0.0 to 1.0 and the nominal times included are 0.1 to 10 seconds. As we can see from the figure, there was no favored client. In terms of the fraction of deadlines met, the performance of all the scheduling strategies was similar, although RANDOM did lag slightly. The point is that the MEASURE and prediction-based strategies were no more prone to catastrophic synchronization than the pure RANDOM strategy. The fact that the prediction-based strategies didn't really do better than the MEASURE strategy here suggests that the primary factor in avoiding contention is that task submission is not synchronized with time or between clients. In terms of the fraction of deadlines met when possible, we can see that, as before, the prediction-based strategies are more trustworthy from the application point of view—when they assert that a deadline can be met, it is likely to be met. Notice that the AR(16) strategy achieves the target 95% level for all the clients. In terms of the randomness introduced by the prediction-based strategies, we can once again see that the AR(16) strategy is able to introduce randomness. Although the average number of possible hosts is small, by the reasoning presented above, it should be sufficient to quickly desynchronize the schedulers if the task submission pattern should happen to synchronize with time or across clients.

6.7 Conclusion

This chapter described the interface of the real-time scheduling advisor and its implementation using the prediction-based running time advisor described in the last chapter. In addition to such prediction-based strategies, we also considered a purely random strategy and a strategy based simply on host load measurement. Because the real-time advisor operates in a shared, unreserved computing environment that it does not control, deadlines can be missed due both to advisor error and to the whims of the computing environment.
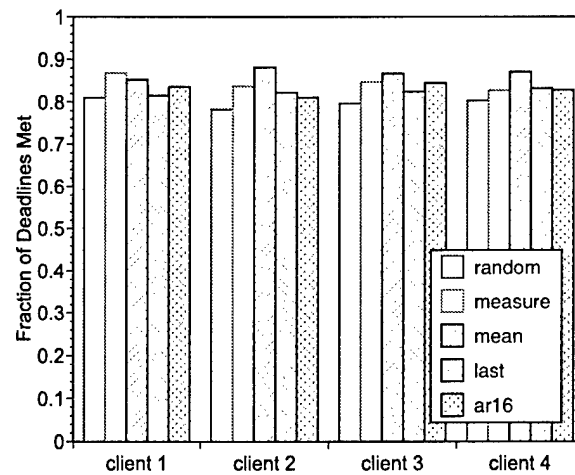
(a) fraction of deadlines met
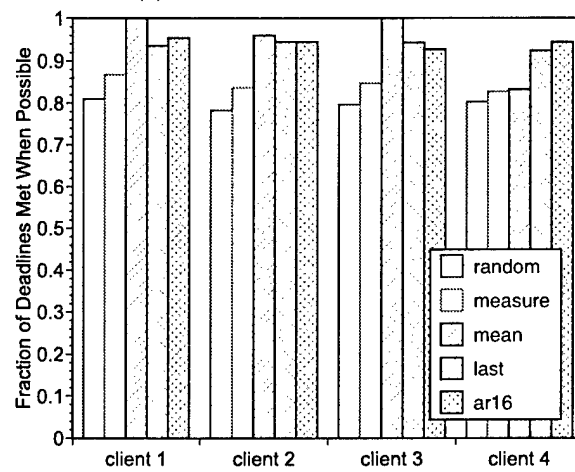
(b) fraction of deadlines met when possible
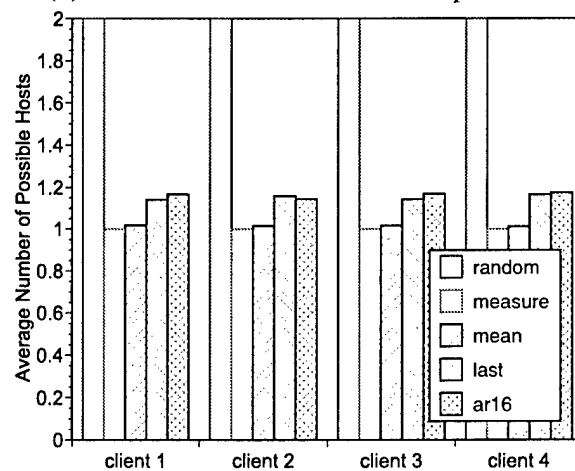
(c) number of possible hosts

Figure 6.18: Scheduling results for multiple contending clients, 0 to 1 slack, 0.1 to 10 second tasks.

In order to better understand the probability of meeting a deadline in a particular environment, and how sensitive that probability is to prediction errors on the part of the advisor, we constructed an analytic model that helped us gain an intuition for real-time scheduling advisor performance, and to guide our understanding of the empirical evaluation of the advisor.

The core of our evaluation was based on measuring how the different strategies actually worked in interesting real (albeit reconstructed) environments. The main conclusion of the evaluation was that the prediction-based strategies, particularly the AR(16)-based strategy, are superior to purely measurement-based and random strategies. The AR(16)-based strategy is able to increase the probability that a deadline will be met over that of the measurement-based strategy. The increase is particularly strong near the critical slack, where deadlines are just able to be met in expectation. Another observation is that, in contrast to the random and measurement-based strategies, the prediction-based strategies can tell the application when they believe the deadline can be met using the host they recommend. AR(16) is the most trustworthy in this respect. When the AR(16)-based strategy tells the application that the deadline can be met, the chances it will actually be met are very high, and, in most cases, the same as the confidence level that the application requested. Finally, unlike the measure-based strategy, the prediction-based strategies are able to introduce significant randomness into their scheduling decisions, which decreases the chance that two or more advisors will disastrously synchronize their actions. Given that all of these benefits are purchased at only a tiny overhead over the measure-based strategy, the case for using the AR(16)-based strategy in real-time scheduling advisors is clear.

Chapter 7

Conclusion

This dissertation has introduced the concept of real-time scheduling advisors and has argued for basing them on explicit resource-oriented prediction, specifically on the prediction of resource signals. To support the utility of such advisors, we identified the characteristics of a class of applications, distributed interactive applications, which can benefit from them, and provided examples of applications in this class. To support basing real-time scheduling advisors on the explicit prediction of resource signals, we showed that, in contrast to designs which are based on application-oriented prediction, a resource-oriented design is more scalable, makes decisions based on more up-to-date information, can support other forms of adaptation advisors, and can easily leverage advances in statistical prediction techniques. However, unlike the application-oriented approach, the resource-oriented approach produces and uses resource availability information that exists at considerable remove from the performance of the application. The core of the dissertation shows that it is nonetheless possible to span this gap.

To show that an effective real-time scheduling advisor can be based on the prediction of resource signals, we designed, implemented, and evaluated a prototype system that uses host load prediction based on high-order autoregressive time series models to schedule compute-bound tasks. The design of that system is illustrated in Figure 7.1. We evaluated each layer of the design. We found that the system provides effective scheduling advice to applications. Furthermore, because the system is resource-oriented and because of the careful compartmentalization of its functionality, it is not merely a real-time scheduling advisor, but can also provide several other kinds of useful information to applications, other forms of adaptation advisors, and other middleware. These include statistical estimates (confidence intervals) for the running time of tasks or for available time on the processor, predictions of the host load signal along with estimates of prediction error, and raw measurements of the host load signal, including histories.

We have also introduced a resource signal methodology for attacking the problem of predicting the availability of a new resource, as well as a toolkit to help carry out the methodology. The ultimate result of using the methodology and the toolkit is a high performance on-line prediction system for the resource. We applied the methodology and the toolkit to determine how to predict the host load signal. The resulting host load prediction system is used as the basis of the real-time scheduling advisor of Figure 7.1 and has also been incorporated in the CMU Remos resource measurement system [82], and into BBN's QuO distributed object quality of service framework [145].

In the remainder of this chapter, we summarize the steps and contributions of this dissertation, discuss related work, and show how we intend to extend the framework of Figure 7.1 in the future.
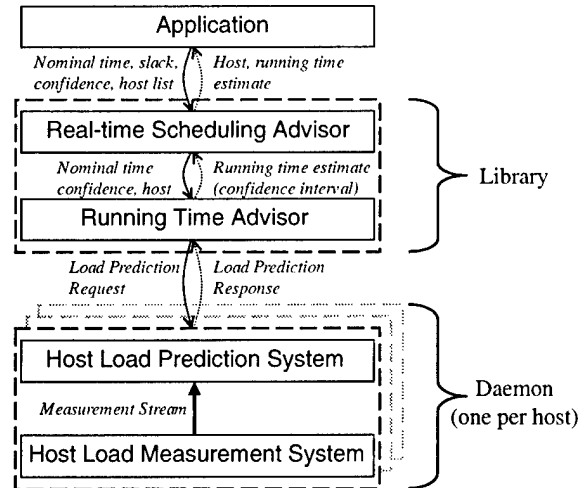
Figure 7.1: The structure of the prototype host load prediction-based real-time scheduling advisor (identical to Figure 1.6).

7.1   Summary and contributions

The following list summarizes the work described in this dissertation and the insights and contributions that it has produced.

- We identified the class of distributed interactive applications. In these applications, computation takes the form of tasks that are initiated by user action, which occurs aperiodically. Tasks are executed sequentially because they provide feedback that determines the user's next action. Because of this interactivity, consistent responsiveness is paramount. This requirement can be expressed in the form of a real-time deadline on each task's running time. However, the applications are resilient in the face of missed deadlines. Furthermore, the applications have been built with distributed operation in mind. When a task arrives, a distributed interactive application knows its resource demands and its deadline, and can choose which host to run it. In addition, it may also be able to adapt the resource demands of the task by changing the computation it performs. (Chapter 1)

- We provided four examples of distributed interactive applications: QuakeViz, OpenMap, Acoustic CAD, and an image editor. We measured the CPU demands of representative QuakeViz applications, which we use as the basis of our randomized evaluations in this dissertation. (Chapter 1)

- We focused on running distributed interactive applications on typical shared, unreserved distributed computing environments. These environments do not provide resource reservations, admissions control, or a global notion of priority. Because of this lack of infrastructure, which seems likely to persist into the foreseeable future, traditional distributed soft real-time systems can not satisfy the needs of these applications. (Chapter 1)

- We introduced the concept of real-time scheduling advisors. A real-time scheduling advisor is a middleware service that advises the application as to the host where a task's deadline is most likely to be met. The advice may be supplemented with additional information, such as the expected running time of the task. However, it is strictly best-effort. In essence, the real-time scheduling advisor guides the application in using its adaptation mechanisms (the choice of host, and perhaps also the ability to change resource demands) to help the task meet its deadline. The combination of these semantics, the

characteristics of distributed interactive applications, and the shared, unreserved distributed computing environments we target defines the scheduling problem that real-time scheduling advisors solve. (Chapter 1)

- We explored the design space for real-time scheduling advisors and concluded that the prediction of resource availability, either implicit or explicit, is the basis of most designs. Explicit prediction approaches can furthermore be divided into application-oriented approaches, in which the performance of the application's tasks are probes on resource availability, and resource-oriented approaches, in which resources are monitored separately from the application. (Chapter 1)

- We identified the tradeoffs between the application-oriented approach and the resource-oriented approach. The resource-oriented approach is more scalable, makes decisions based on more up-to-date information, can support other forms of adaptation advisors, and can easily leverage advances in statistical prediction techniques. However, unlike the application-oriented approach, the resource-oriented approach produces and uses resource availability information that exists at considerable remove from the performance of the application. If this gap between the resources and the application can be spanned, then the resource-oriented approach is preferable. The core of the thesis shows that it can indeed be spanned. (Chapter 1)

- We explored the performance of the application-oriented approach, developing a predictive algorithm that provides near-optimal performance in simulation for a simplified version of the scheduling problem. The disadvantage of the application-oriented approach is not in its performance, but in its scalability problems. (Appendix A)

- We defined the notion of a resource signal, which is a scalar-valued, discrete-time signal that correlates with the availability of a resource. The resource we focused on in the thesis is CPU time. The resource signal we used is host load, specifically, the Digital Unix five second load average. (Chapter 1)

- We designed the framework for a real-time scheduling advisor (Figure 7.1) that is based on the prediction of host load signals. The core of the thesis fills out the structure shown in the figure. The framework can be extended to include other resources and other forms of advisors, as we describe later in this chapter. (Chapter 1)

- We developed a methodology for attacking the problem of predicting resource availability. The main idea behind the methodology is to transform a specific resource prediction problem into a general time series prediction problem as early as possible, and then to apply the substantial statistical machinery that already exists to address such problems. We applied this methodology to develop the host load measurement and prediction systems shown in Figure 7.1. Later in this chapter, we show how we are beginning to apply it to predicting the bandwidth of network connections. (Chapter 2)

- We designed, implemented, and evaluated the RPS Toolkit, which helps to carry out the latter steps of the methodology. The early steps of the methodology, which are done off-line and which are human-intensive, are well served with existing tools. The latter steps, which are concerned with large scale, machine-intensive evaluation, and building efficient on-line prediction systems, were, however, not well served. RPS improves this situation greatly. RPS consists of sensor libraries, a time series prediction library, a communication library, prediction components, and a parallelized evaluation system. The prediction components are composed at run-time to form on-line resource prediction systems. The overheads of these systems, when run at the measurement rates we typically expect to use, are miniscule. Furthermore, they are capable of sustaining measurement rates 2-3 orders

of magnitude higher than we typically need. RPS-based prediction systems have been incorporated into the CMU Remos [82] and BBN QuO [145] systems. (Chapter 2)

- We identified host load, specifically, the Digital Unix five second load average, as being an appropriate resource signal for CPU availability. We determined that 1 Hz was the appropriate rate at which to sample this signal. (Chapter 3)

- At two different times of the year, we collected week-long, 1 Hz traces of host load signals on a large number of different machines that we classify as production and research cluster machines, compute servers, or desktop workstations. These traces form the basis of our studies and evaluations in this dissertation. (Chapter 3)

- We developed a technique called load trace playback which recreates a facsimile of the workload that a load trace measured. We used this technique to evaluate the running time and real-time scheduling advisors in Figure 7.1. We have also provided the playback tool and our collection of load traces to the research community as a way of producing realistic workloads. (Chapter 5).

- We performed a detailed statistical study of the traces to determine the prospects for predicting host load signals. In particular, we focused on those aspects of the signal that are pertinent to prediction. This is the first study of this kind of which we are aware. We found that host load signals exhibit substantial autocorrelation structure, far beyond that which one would expect from merely the exponential smoothing the operating system applies to the signal. This suggested that linear models, which attempt to model such autocorrelation structures parsimoniously, might be appropriate for host load prediction. (Chapter 3)

- We found that the host load signal is self-similar. This new result suggested that more complex and expensive linear models, such as ARFIMA models, might be required to predict these signals. (Chapter 3).

- We found that the host load signal exhibits epochal behavior—the signal remains stationary for an extended period of time, and then abruptly transitions to another stationary regime. This new result argued against linear time series models because even those that explicitly model nonstationarity cannot model such abrupt transitions. Furthermore, if linear models could be used within an epoch, it might be necessary to refit such models at epoch transitions. (Chapter 3)

- Having decided, with reservations, that linear models might be appropriate for host load prediction, we performed a large scale, randomized evaluation based on the traces to determine which kind of linear model, if any, was most appropriate. The study was unique because of its large scale, randomized approach, fine grain prediction, and focus on more sophisticated models. The surprising conclusion is that not only are linear models appropriate, but that relatively simple linear models are sufficient to provide useful predictions as far as 30 seconds into the future. We concluded that autoregressive models of order 16 or better (AR(16)) are appropriate for host load prediction. More sophisticated models have similar predictive power while requiring considerable more CPU time to use. (Chapter 4)

- We implemented and evaluated the performance of an RPS-based host load prediction system that can be configured to use any predictive model. We use this system to evaluate the running time and real-time scheduling advisors of Figure 7.1. In addition to AR(16) models, we also used the LAST model (last sample as the prediction of all future samples), and MEAN model (long-term average of the signal as the prediction of all future samples) in our evaluations. For each of these models, the host load prediction system has negligible overhead. (Chapters 4, 2)

- We developed an algorithm to transform from host load predictions and a task's CPU demands to a confidence interval for the task's running time on the host. This forms the basis for the running time advisor shown in Figure 7.1. The transformation effectively models the Unix scheduler from the perspective of a new task. The host load predictions include estimates of prediction quality, namely the covariance matrix of the prediction errors, which form the basis of computing the confidence interval. In addition, for small tasks, it is vital to model the priority boost that the scheduler provides processes that have just finished an I/O operation such as returning from a read on a socket. We introduced a pre-processing step called load discounting to account for this effect. (Chapter 5)

- We evaluated the running time advisor by running a large number of randomized testcases on hosts whose workloads were played back from the load traces we collected. The main result is that our system, using a strong host load predictor such as the AR(16) predictor, does indeed compute reasonable and useful confidence intervals for the running time of tasks. In comparison to LAST and MEAN, the benefits of AR(16) depend on the how heavily loaded the host is and on the nominal time of the task. On heavily loaded hosts, AR(16) models produce appropriately wider confidence intervals that correctly capture the increased variability in running time. On lightly loaded hosts, AR(16) models produce narrower confidence intervals that still cover the desired fraction of tasks. (Chapter 5)

- We developed an algorithm to select an appropriate host for meeting a task's deadline given confidence intervals for the running time of the task on different hosts. The algorithm attempts to introduce as much randomness as possible into the host selection process while still choosing a host on which the deadline will be met with the user's specified probability. This algorithm is the basis for the real-time scheduling advisor component of Figure 7.1. (Chapter 6)

- We developed an analytic model for the performance of prediction-based real-time scheduling which shows how scheduling feasibility and predictor sensitivity depend on different parameters of the set of hosts to which tasks can be scheduled (the scenario). This model provides a framework for accounting for how deadlines are missed, and helps us explain the results of the empirical evaluation of our real-time scheduling advisor. (Chapter 6)

- We evaluated the performance of the real-time scheduling advisor by running a large number of randomized testcases on hosts whose workloads were played back from the load traces we collected. We compared the advisor to two other approaches: selecting a host at random, and selecting a host whose host load was measured to be the lowest. The measurement-based approach performed considerably better than the random approach in terms of the probability that a deadline would be met. In comparison to measurement, the prediction-based real-time scheduling advisor always performed at least as well by this metric, and considerably better in resource constrained situations. When there are just sufficient resources to meet a deadline, the prediction-based advisor significantly increases the probability that it will be met. In addition, the prediction-based advisor informs the application of the confidence interval for the task's running time on the host it has chosen. This means that the application knows, with considerable confidence, whether the deadline will be met *before* it runs the task, enabling it to adapt the task's CPU demands or deadline, if necessary. This is a capability that does not exist with the purely measurement-based or random scheduling approaches. Finally, the prediction-based approach is able to introduce considerable randomness into its scheduling decisions while providing the application with good performance. This means that it is much less likely to interact badly with other advisors running in the system. The additional benefits over the purely measurement-based approach come at a miniscule additional cost given the simple and cheap AR(16) model and the low overheads of RPS. (Chapter 6)

- The design, implementation, performance, and efficiency of the prototype host load prediction-based real-time scheduling advisor show that it is feasible to base real-time scheduling advisors on the explicit prediction of resource signals. Furthermore, we have shown that these advisors can provide a useful service to distributed interactive applications.

7.2 Related work

The work described in this dissertation is related to work in a number of other areas.

7.2.1 Applications

There is considerable interest in distributed interactive applications for visualizing and steering scientific computations, accessing replicated remote data, manipulating media, and playing games. These kinds of applications must provide responsive behavior for their users, a requirement that can easily be expressed as deadlines on their tasks. The goal of the thesis work is to provide services upon which developers of these applications can rely to provide responsiveness. The introductory chapter introduced four applications: the Dv framework for distributed scientific visualizations, BBN's OpenMap framework for presenting cartographical information, Acoustic CAD, and image editing.

Scientific visualization involves the useful display of large, complex datasets, while computational steering lets the user use this display to focus a running physical simulation on areas of interest [58]. For example, the CAVE project [36] involves connecting immersive virtual environment simulators to remote physical simulations. CAVE has also proven to be a good environment for scientific and engineering collaboration [110]. There is increasing interest in developing distributed versions of such systems. The CAVE researchers have run visualizations over the wide area on dedicated machines and networks. Knittel presented an algorithm to parallelize volume visualization on a dedicated workstation cluster [71]. The goal of the Dv project is to build a framework for high performance scientific visualization on common shared, unreserved distributed computing environments [3]. Cumulus [52] is a library and run-time system for adding visualization and computational steering to simulations based on iterative operations on dense matrices. The Acoustic CAD application we outlined in the first chapter is effectively a computational steering application which implements auralization [14] through physical simulation instead of geometric approximation. Tele-operation is related to computational steering, except that the remote physical simulation is replaced with an actual physical system. Tele-operation systems such as the one discussed in [94], involve operating robots by remote control over conventional networks.

Popular information sources are often replicated on multiple servers. This introduces a challenge to the client: which server will provide the best performance? This common problem on the World Wide Web has seen considerable research and proposals [102, 120, 93]. Outside of the Web context, another example is BBN's OpenMap framework [13, 70], which integrates cartographic information from multiple sources, each of which may be replicated. The problem is obviously very similar to the problem of choosing which host is most appropriate to run a real-time task, which this dissertation addresses. The primary difference is that the servers are usually distributed over a wide area, making communication performance much more important.

While most current distributed multimedia systems involve communicating and synchronizing different audio and video streams, manipulating media on a distributed system in computationally significant ways is becoming increasingly important. For example, medical imaging systems [107], involve acquiring, processing, storing, and making medical images easily available to physicians no matter where they are. Next generation stream-oriented systems like the VuSystem [79] filter the streams in computationally intensive ways, such as extracting titling text from video streams. Image editing programs such as Photoshop [2]

are also ripe for distributed systems because of the increasingly huge size of photographic images, as we discussed in the introductory chapter.

Games are becoming increasingly interesting applications for distributed systems. For example, the Department of Defense's DIS effort is trying to create large scale (100,000 users or more) simulated war games by connecting different kinds of simulators located all over the world [35]. At least three companies, Silicon Graphics, Zombie Entertainment, and Military Simulations, Inc., are working on extending the emerging DIS technology for civilian games, with an example of a DIS-based game being shown at SIGGRAPH '96. In addition to multiplayer games, we expect that single player games will increasingly rely on computationally intensive physical simulations for realism. For example, the use of physically based acoustics and music [4, 64, 126] in games or design applications such as Acoustic CAD may not be far off.

7.2.2 Remote execution

The purpose of our work is to decide which host is best for a given task. We assume that the application can run a task on more than one host. This requires a remote execution facility, of which many examples exist. Examples of remote execution facilities include remote procedure call systems, distributed shared memory mechanisms, and distributed object systems. Remote procedure call systems [20, 17], such as the one specified by the DCE [131] standard, provide a procedure call-like abstraction over network communication with a remote server. Distributed object systems [101, 28], such as CORBA [134, 98, 123], DCOM [26] and Java RMI [129], extend the RPC abstraction to objects by allowing the association of state with a group of procedures. Distributed shared memory systems such as Tempest [111], Shasta [116], and TreadMarks [37] provide the appearance of a global address space shared by some number of private memory hosts. This is combined with a multithreading model where threads that may be assigned to different hosts communicate via shared variables. Although today's common remote execution facilities have relatively high overheads in the millisecond range [54], our experience is that this is due mostly to the latency of network protocol stacks and we expect that this latency will improve. With microsecond range application-to-application communication latencies such as the PAPERS network [62] can provide, the overhead of remote execution in a system like CORBA or Java RMI could be within a couple of orders of magnitude of the overhead of a local call. For example, the marshalling/unmarshalling and dispatch overhead of a call in our LDOS distributed object system is only about 100 times greater than that of a C++ virtual function call. Such improvements will drastically increase the fraction of tasks in a typical program that could be profitably executed remotely.

7.2.3 Dynamic load balancing

Dynamic load balancing involves assigning tasks to hosts at run-time or moving tasks from one host to another at run-time to maximize some performance metric. Common performance metrics include the throughput of the system, the mean execution time of tasks, or the actual running time of an application. In contrast, the goal of our work is to maximize the probability that an individual task will meet its deadline. However, it is important to note that the running time advisor component of our work can be used in pursuit of other goals, such as those of load balancing. There are both operating system-centric and application-centric approaches to dynamic load balancing. Our work is wholehearted application-centric—we predict the behavior of the rest of the system to improve the performance of one specific application. However, we share an important question with both forms of dynamic load balancing: how can an individual host in a distributed system be aware and predict the behavior of the system as a whole in a scalable, practical way?

The goal of operating system-centric dynamic load balancing is usually to maximize the throughput of the distributed system as a whole. The idea is to schedule independent, sequential tasks on a group of hosts such that each host has approximately the same load. A distinction is sometimes drawn between load

sharing [27] (also called load distribution and load leveling), which offers only a rough approximation to equalizing load across the hosts, and load balancing, which attempts more precision [39]. In either case, the distributed operating system is comprised of host-local schedulers which interact to implement transfer (should task be moved?) and location (where should task go?) policies [38]. In our work, we assume that the host-local schedulers act independently and that we have no control over their decisions. We attempt to predict how they will respond to the task that we desire to introduce. Furthermore, we are unconcerned about balancing load.

Early work in operating system-centric dynamic load balancing assumed simple, analytically tractable distributions for job length and other properties. Under these assumptions, simple heuristics and initial placement were found to be adequate [40, 39]. However, measurement studies [77] have cast doubt on these assumptions, and more recent work [60] argues strongly for process migration. This is an important result for our work, since it argues for changing the mapping of the application as it runs, which we do on a task by task basis.

The goal of application-centric dynamic load balancing is to minimize the execution time of a single, typically parallel application. For example, in data parallel applications built in DOME [6], neighboring hosts periodically exchange measurements of execution time and redistribute their data accordingly. The system described in [124] uses a global approach where one centralized agent makes load balancing decisions based on execution times reported by all of the hosts. Jade [113] load balances unfolding task parallel computations by dynamically mapping task graph nodes to hosts to minimize communication and overall execution time. In contrast, our work dynamically maps sequential real-time tasks. However, it important to point out that the running time advisor component of Figure 7.1 is not specific to real-time tasks and could be used by an application-centric dynamic load balancer.

To be scalable, a dynamic load balancing system pursues its global policy by making local decisions based on limited or outdated knowledge of the system as a whole. Based on this limited knowledge, a local scheduler estimates the current state of the system and makes its decisions based on that estimate. By exploiting subtle properties of specific distributed systems or application workloads, some systems can put their limited knowledge to better use. In essence, this is what we do for a specific class of applications. One example of exploiting such properties is Hailperin's thesis [59], which takes advantage of the statistical periodicity of his sensor-driven applications. In contrast our applications present aperiodically arriving tasks. In [122], the authors present location policies that adapt to system load to avoid instability. Mehra and Wah [88] use comparator neural networks to predict current workload indices on remote hosts using outdated information about resource utilization. Mehra's thesis [87] describes a whole load balancing system based on automated strategy learning using comparator neural networks. We have also found some success with a neural networks approach (see Appendix A).

7.2.4 Distributed soft real-time systems

A real-time system allows a programmer to specify a deadline for the execution of a one-time or periodic task. In this sense, a real-time scheduling advisor is a real-time system. However, while real-time systems typically provide some form of guarantees, our work is strictly best effort. Furthermore, most real-time systems require control of the entire computing environment, either via resource reservation or priority-based scheduling with globally observed priorities. In contrast, we control only the mapping of individual tasks to hosts. As far as we are aware, the idea of an entirely application-level real-time scheduling advisor that provides scheduling advice on a best-effort basis is unique to this dissertation. We described the idea of such a tool and argued for basing it on prediction in an earlier document [29].

A hard real-time system is one which is only correct when every task in the system always meets its deadlines. These systems are necessary for some applications, but are very difficult to build, place many

constraints on the hardware, and typically require knowledge of all tasks at design time. The rate-monotonic scheduling and earliest deadline first scheduling algorithms [81] can be used to produce feasible schedules for fixed sets of tasks with known characteristics. More recent work has extended queueing theory to reason about real-time tasks in queueing and queueing network systems [75, 76]. Distributed hard real-time systems have been built (eg, MARS [119]), but these tend to rely on specialized hardware. In contrast, our work offers only a best effort service, but the tasks and deadlines are discovered only as the program is executed.

A soft real-time system is one where it is permissible for some tasks to miss their deadlines. What it means to miss a deadline depends on the system. A common approach is to generalize a deadline into a utility function of the duration since the task arrival time [68, 67], or as utility function that includes timeliness as well as additional independent variables [114]. The system then attempts to maximize the collective or individual utilities of tasks in the system. The functional form of a hard deadline in this kind of system is a unit step function. While the general case of this scheduling/optimization problem is quite complex, by constraining the form of the utility function, tractable specialized scheduling algorithms can be developed.

A common underlying mechanism in soft real-time is a fixed priority scheduler combined with priority inheritance to address priority inversion. In a fixed priority scheduler, each task is assigned a fixed priority and the scheduler always runs the highest priority task that is ready to run. Another common mechanism is that of resource reservation and admission control, such as in resource kernels [108]. Unfortunately, very few general purpose operating systems actually implement fixed priority scheduling, or provide resource reservations and admission control. In contrast, we have a very simple performance metric (the fraction of tasks that meet their deadlines) and our work does not require specific kinds of schedulers or any real control over the system other than a remote execution facility and the ability to measure.

Achieving soft real-time goals in a distributed system is very difficult since the system is much larger and more complex. One approach is to create a notion of a global priority, require that all resources in the system be scheduled by fixed priority schedulers that respect the fixed priority, and require that all communication delays be bounded. This is the approach taken in [139] and in proposals to the Real-time CORBA standardization effort [97, 34]. Another approach is to implement resource schedulers that provide reservation mechanisms, such as those in SONIC [104] or the Resource Kernel [108], and then rely on end-to-end resource reservations. In contrast, we make no attempt to control the system to meet our goals. Rather, we monitor and predict the system to adapt the application's behavior (the mapping of tasks to hosts) to the system in order to meet our deadlines.

As in dynamic load-balancing, a key challenge in distributed soft real-time systems is how to scalably coordinate the actions of many local schedulers. Indeed, a distributed real-time system can be viewed as a load-sharing system with real-time tasks [74]. If a task is submitted at a host where it cannot meet its deadline, we would like to move it to a host where it is likely to be able to. However, that host's knowledge of other hosts is limited and out of date. Bestavros and Spartiotis suggest a scheme in which simple stochastic models of other hosts' load conditions are formed based on infrequent, opportunistic information exchanges and task mappings are chosen probabilistically from among hosts that qualify according to the models [19]. Bestavros extends this work by arguing that load balancing is actually detrimental to real-time performance and develops an alternative which intentionally tries to keep loads out of balance and makes task mapping decisions based on the underlying load profile it is trying to maintain [18]. We suspect that Bestavros has more success with simple stochastic models than we do (in Appendix A) because he is modeling the remote schedulers that are a part of his system and are therefore a known quantity. In addition to load balancing, Hailperin's thesis [59] also takes advantage of the statistical periodicity of his sensor-driven applications to help place and migrate objects so that method invocations meet their deadlines. We must deal with aperiodically arriving tasks.

7.2.5 Quality of service

Quality of service, or QoS, is a fairly generic, but nonetheless widely used term to describe providing a more predictable environment for applications. In this sense, our work can be seen as a QoS service for the specific class of distributed interactive applications. In the networking community, QoS has meant providing deterministic and statistical guarantees of the bandwidth and latency of connections for media applications [44, 73]. Tenet [45], RSVP [24] and the ATM CBR and VBR service classes are examples of network QoS systems that are based on reservations and admission control.

Quality of service also includes the notion of adaptive applications [95], or application-level scheduling [16]. The idea is that the application can modulate its behavior in response to changing resource availability in order to provide graceful degradation of the user experience. Our work relies on one such adaptation mechanism: the ability to choose which host will run a task. In addition, the real-time scheduling advisor can provide estimates of task running time to the application, which enables it to change its resource demands via other adaptation mechanisms.

Recently, interest has focused on how to expose network and host QoS features to the application programmer in an understandable fashion. For example, the QuO project at BBN [145] is considering how to express QoS requirements and guarantees for distributed objects. The challenge is in translating between the object level and the network and host levels and in providing a clean way for the programmer to identify different regions of operation and when to switch between them. QuO's adaptation mechanism is based on routing and translating CORBA method invocations. We have incorporated our host load prediction system in QuO. Another example of translation and adaptation involves the Qual language and QosME environment developed at Columbia [47]. Qual is a language for expressing the quality requirements of program modules. These specifications can then be compiled into a monitoring agent which switches between different regions of operation. Compilable specification languages to support binding modules and services have also been developed in the parallel computing community. DURRA [10] is a well known example. In our work, the deadline placed on the running time of a task can be seen as a QoS specification, and the choice of host is an adaptation mechanism.

The distributed soft real-time systems community is exploring QoS as a way of extending the notion of a utility function on timeliness with other dimensions. For example, utility functions based on precision, accuracy, and timeliness have been suggested as a more appropriate abstraction [114]. The idea is to expose the fact that many applications can tradeoff between the tardiness of a computation and the quality of its results. The Q-RAM model [109] extends these ideas to arbitrary dimensions and provides a way to reconcile the QoS tradeoffs of a group of applications to maximize a global quality metric. Amaranth [5] incorporates Q-RAM, real-time queueing theory, and application adaptation to provide probabilistic QoS guarantees.

In mobile computing the QoS challenge is to detect changing wireless network conditions quickly and adapt to them gracefully. For example, the Odyssey system [95] provides the application with typed data streams (eg, video) and switches between different quality levels (data rates) as it detects changing available network bandwidth. When quality strays outside of the range that the application registered initially, an upcall is made to the application, which decides the course of action.

7.2.6 Resource measurement and prediction systems

RPS is a toolkit for building on-line resource measurement and prediction systems [32]. Several other resource measurement systems also exist, one of which is also a prediction system. An essential prerequisite for our work is the ability to sample the resource signals in a distributed computing environment. In addition the RPS's sensor libraries, these other resource measurement systems can also produce measurement streams that RPS-based systems can use.

Remos [82] is an interface by which applications can query for information about the network and the

hosts attached to it. The answers to application queries are in the form of graphs that reflect the logical topology of the network. The graphs are annotated with dynamic information such as bandwidth. Applications can also pose queries about the bandwidth and latency of flows (ie, network connections) through a graph. The current Remos implementation integrates information about local area networks collected via SNMP with information about wide area networks collected by measuring the performance of small data transfers (benchmarking) to answer application queries [89]. The RPS-based host load prediction system described in this dissertation provides host measurements and predictions for Remos. RPS's network flow bandwidth sensor, described in Chapter 2, is based on Remos flow queries. We are currently integrating RPS more deeply with Remos with the goal of providing prediction services for arbitrary resource signals.

Topology-d [99] is another example of a network measurement system that can provide measurement streams which could be converted to prediction streams via RPS-based systems. Like Remos, Topology-d is based on the notion of logical topologies. However, it infers them differently. SPAND [120] passively observes application traffic in order to evaluate network performance. Bolliger, et al developed software to monitor the performance of TCP connections at the packet level [22].

The Network Weather Service [142, 141, 140] (NWS) provides measurements of network bandwidth and latency, as well as of host load. On the host side, NWS includes a unique sensor that incorporates both the load average signal, which measures the contention for the processor, and the output from the Unix vmstat utility, which measures the utilization of the processor. The system monitors how well each signal correlates with the running time of a sample process and then dynamically chooses the better signal. On the network side, NWS is based on benchmarking.

The Network Weather Service is the only other example of an on-line resource prediction system we are aware of. NWS can use windowed mean, median, and AR filters to predict the network and host signals that it produces. RPS's repertoire of models is considerably larger. NWS is a production system that tries to provide a ubiquitous resource prediction service for metacomputing. In contrast, the aim of RPS is to be a toolkit for constructing such systems and others. The RPS user can commit to as little or as much of RPS as is desired. We believe NWS and RPS are complementary. For example, RPS-based systems could use NWS sensors, or NWS could use RPS's predictive models.

Interactive data analysis tools such as Matlab [86, 85] and S-Plus [84] provide many of the statistical and signal processing procedures needed to study resource signals and to find appropriate predictive models. Ideally, large scale, randomized evaluations of such candidate models are then needed. The RPS-based parallelized evaluation system we used in this thesis helps to efficiently carry out such studies.

7.2.7 Workload characterization

Considerable effort has gone into characterizing workloads of hosts in distributed systems. However, the study described in Chapter 3 ([30]) is the first study we are aware of that explores the behavior of the Unix load average at fine timescales and its implications for prediction. It is also the first to discover that the host load signal is self-similar and that it displays epochal behavior. Wolski, et al have confirmed the self-similarity result [141].

Almost all prior work has been from the perspective of operating system-centric dynamic load balancing and sharing, as described above. The main issue in this context is the distribution of job sizes and its implications for process migration. Early measurements suggested that job sizes were exponentially distributed, which implied that process migration was unnecessary [40]. This assumption continued to be used despite a measurement study by Leland, et al, who found that the distribution actually followed a power law [77]. It was only recently that this result was confirmed and its implications fully explored [60]. An important implication for our work is that power law distributions suggest that process migration is beneficial, which we certainly do on a task by task basis. That job size follows a heavy-tailed distribution helps to explain the

self-similarity of the host load signal.

Mutka and Livny's study [92] is perhaps closest to the work of Chapter 3. They studied workstation availability as a binary function of load average and other measures. The point was to determine how much idle CPU time could be reclaimed by the Condor system [80], and to what extent the intervals during which Condor could appropriate the CPU could be predicted. The authors developed a simple Markovian model for their on-off notion of CPU availability which provided a certain degree of predictability. In contrast, our work observes a continuous measure of CPU contention in order to estimate the slowdown a task will likely encounter on it.

There has been considerable work in characterizing network workloads. This work has largely been focused on finding appropriate, analytically tractable models for queueing analysis, and on models to generate realistic network traffic for simulation work. Interestingly, just like the load balancing community, the networking community also made the mistake of initially choosing an analytically convenient models (Poisson arrivals) that ultimately proved to be utterly wrong [103]. In general, the importance of self-similarity [78, 137, 138] and multifractal properties [43] in local area and wide area network traffic has become increasingly clear. The existence of these properties meant that network switches needed much deeper buffers to handle traffic bursts than previous analytic work had suggested. They have also resulted in new models to generate traffic for simulation work [137, 112]. However, at this point, their implications for network *prediction* are unclear. On a bright note, Balakrishnan, et al's study [8] of traffic to an Olympics web site did find performance correlations that could be beneficial to prediction.

The difficulties that both the load balancing and networking communities have had in finding appropriate workload characterizations pushed us in the direction of using real workloads to evaluate our system. The host load trace playback technique we developed is in the spirit of trace modulation, a technique that has been very useful in the mobile networking community [96].

Bassingthwaighte, et al [11] provide a good intuitive introduction to self-similarity and Beran [15] provides a good introduction to long-range dependence.

7.2.8 Studies of resource prediction

Although there has been a vast amount of work on studying and modeling workloads, which we have touched on in the previous section, very little of this work has focused on prediction from the point of view of applications. This is changing as application developers increasingly target complex, shared distributed systems and research continues into building shared computational grids [48].

The earliest work on predicting host load is that of Samadani and Kalthofen [115]. They found that small ARIMA models were preferable to single-point predictors and Bayesian predictors for predicting load. Their empirical study concentrated on coarse grain prediction (one minute sampling interval) of four traces that measured load as the number of non-idle processes shown by the Unix "ps" command. In contrast, we studied finer grain (one second sampling interval) prediction of the Digital Unix five-second load average on a much larger set of machines using higher order models as well as the ARFIMA class of models. Additionally, our study was randomized with respect to models instead of using the Box-Jenkins heuristic identification procedure. Finally, we reached a different conclusion for our regime, namely that AR models are sufficient.

Wolski, et al have used the Network Weather Service (described above) to study the prediction of network bandwidth and latency [140] and CPU availability [141]. The work on CPU availability, which is contemporaneous with ours [31], used windowed mean, median, and AR filters to predict various measures of CPU load, including the load average. Wolski, et al's finding that simple models such as AR models worked well agrees with our results. Our study furthers this shared result by showing that more sophisticated predictive models are not necessary. One remarkable finding of Wolski, et al's study was that, in terms

of predicting the running time of a probe task using a host load signal (their sensor is different from ours), the prediction error was actually significantly less than the measurement error. We have also found that translating from a host load signal to a running time estimate is a surprisingly difficult task. The implication seems to be that we need more or better measures of CPU availability.

The Network Weather Service was first applied to predict the bandwidth and latency of network connections, using windowed mean, median, and AR filters [140]. The results seem somewhat mixed. On the one hand, the predictive models seem to be very useful in denoising the latency signal. On the other hand, the mean square error levels produced by predicting bandwidth are in many cases not significantly lower than the raw variance of the signal. We have found similar results in a small study of network bandwidth prediction, reported in Section 7.3. There is plenty of work left to be done in understanding and predicting network bandwidth. It is important to note that even if the predictability of a resource signal is limited, a predictive model is extremely useful because it can characterizing the signal's variability.

In addition to Wolksi, et al's work, some other results have been reported in modeling network traffic using linear time series models. Groschwitz and Polyzos attempted to model the long-term growth of traffic volume on the NSFNet backbone using ARIMA models [57]. Basu, et al explored the statistics of modeling shorter term traffic using ARIMA models but did not use them to predict behavior [12].

The AR, MA, ARMA, and ARIMA models have long histories dating back to the 19th century. Box, et al [23] is the standard reference and Brockwell and Davis[25] provide a useful example-driven approach. ARFIMA models are a relatively recent innovation [63, 55].

7.2.9 Application-level scheduling

The running time and real-time scheduling advisors support a form of application-level scheduling [16], in which applications schedule themselves, adapting to the availability of resources. This dissertation is the first attempt to implement application-level real-time scheduling of which we are aware. Other work has focused on throughput-oriented parallel applications such as gene sequencing on metacomputers [127].

In effect, our running time advisor estimates the amount of contention that a task is likely to encounter when run on a particular host. The estimate includes the variability of this contention. Figueira and Berman have studied how to model the effects of contention variability on parallel applications [46].

The running time advisor's estimate is expressed in the form of a confidence interval, which makes it appropriate input for stochastic scheduling[117]. Stochastic scheduling is a framework for reasoning about scheduling parallel computations in environments where resource availability is measured using stochastic instead of deterministic values. The running time advisor component of our system could provide such values.

7.3 Future work

The structure of the real-time scheduling advisor, as shown in the Figure 7.1, provides a framework for the future work described below. This dissertation has developed effective real-time scheduling advisors for compute-bound tasks. We intend to generalize the system so that it can also support tasks with substantial communication requirements. The approach is to apply the resource signal methodology to signals that correspond to the availability of network bandwidth and latency, developing prediction systems for these signals, and then to incorporate their predictions into the real-time scheduling advisor. Over the longer term, we want to consider the prediction of other resource signals and the development of other kinds of adaptation advisors. The following describes our intentions at each of the levels in the figure, from bottom to top.
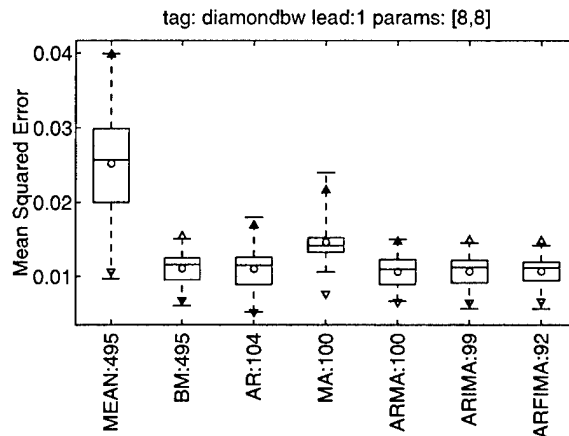
Figure 7.2: An evaluation of linear models for one-step-ahead (approximately 16 seconds) prediction of wide area available network bandwidth measurements between diamond.cs.ucsb.edu and antares.cis.ksu.edu.

7.3.1 Resource signals

We plan to apply the resource signal methodology and RPS to identify, sample, characterize, and discover ways to predict new resource signals. Signals that seem particularly relevant to applications include a network connection's available bandwidth and latency, a host's available memory, and a disk's bandwidth. Being able to predict network bandwidth, for example, would greatly increase the kinds of tasks the real-time scheduling advisor could schedule. The on-line prediction of such resource signals for the benefit of applications is a wide open research area.

We have already begun to work with the bandwidth of network connections. While there are a number of excellent tools, such as Remos [82], for sampling this signal, three interesting problems that have not yet been fully addressed: how do we determine the band limit of the signal, how do we enforce a band limit during the sampling process, and how do we match the almost necessarily non-periodic nature of the sampling process with the periodic requirements that most signal analysis and prediction tools have?

We need to determine the band limit to understand how much of the actual variability of the network bandwidth is hidden during the sampling process. This hidden variability is measurement error which the application or prediction system needs to know about. The sampling process needs to be able to enforce a band limit in order to satisfy Nyquist's theorem and thus avoid sampling artifacts. Finally, the sampling process will likely be non-periodic, especially for wide area networks. However, most signal analysis and prediction tools assume periodic signals. To use them, we will need to determine how to appropriately resample to periodicity.

While no one knows how to answer these questions yet, we have already applied parts of our resource methodology to explore the prediction of some extant network bandwidth signals. For example, we have used RPS to perform small randomized evaluations of the predictive power of linear models on the network bandwidth traces used to evaluate the Network Weather Service [140] (NWS), as well as on self-similar Ethernet traces collected at Bellcore [137]. Rich Wolski and Murad Taqqu helpfully provided the three traces we describe below.

Figure 7.2 shows the results of an evaluation on an NWS trace, which measured the available bandwidth every 16 seconds on a connection between a machine at the University of California, Santa Barbara and a machine at Kansas State University for 64K TCP transfers. The 500 testcases, which used 8 parameter versions of the predictive models for one-step-ahead prediction, are presented using the Box plot format
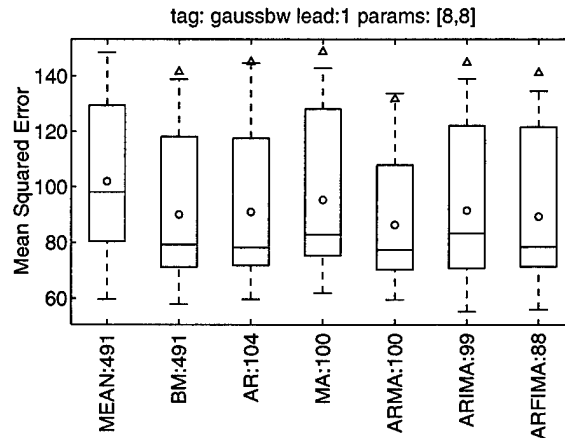
Figure 7.3: An evaluation of linear models for one-step-ahead (approximately 13 seconds) prediction of local area available network bandwidth measurements between gauss and galois, two servers at the San Diego Supercomputing Center.
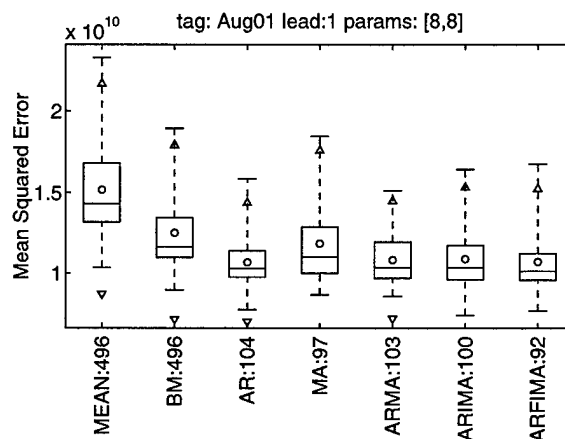
Figure 7.4: An evaluation of linear models for fine-grain one-step-ahead (approximately 100 millisecond) prediction of consumed network bandwidth on a 10 Mbps ethernet at Bellcore.

described in Chapter 4. Obviously, there is some degree of predictability in this signal that can be exploited.

In contrast, a similar evaluation for a local area trace (also from the NWS traces) shows very different results. As we can see from Figure 7.3, this trace, taken between two machines on a local area network at the San Diego Supercomputing Center, does not appear to be predictable with any of the linear models. This begs an interesting question: Are wide area networks more predictable than local area networks?

Perhaps local area networks are simply more predictable on smaller timescales. For example, Figure 7.4 shows the results of an evaluation on a fine-grain trace collected on the Bellcore Ethernet. The original trace consisted of the timestamps and sizes of individual packets, which we integrated to produce a signal of the consumed bandwidth on the network, sampled at 100 millisecond intervals. At this granularity, the trace appears to have some degree of predictability.

7.3.2 Prediction approaches

Some resource signals are unlikely to be predictable using the linear models that are appropriate for host load. For example, networks are well known to exhibit self-similar [137], and even multifractal behavior [43]. The best current model for generating (as opposed to predicting) network traffic uses a wavelet-based model [112]. The combination of this work and our experience with network bandwidth prediction, described above, suggests that more sophisticated predictive models are necessary.

We plan to explore and invent new approaches to predicting recalcitrant signals such as network bandwidth. We are currently considering several interesting approaches that we believe can be adapted for on-line prediction. The first approach is threshold autoregressive models [132], which are designed to model the abrupt transitions between different (linearly) predictable regimes that many resource signals exhibit. The second approach is to model the signal as emerging from a chaotic system of differential equations, an approach which can recover multiple dimensions of behavior (eg, multiple hosts driving a network) flattened into a single signal [1]. The third approach we are considering is wavelets, which simultaneously capture time domain and frequency domain behavior [143]. For example, they would be ideal for capturing the epochal behavior we identified in host load signals. Although wavelets have been pressed into service for *synthesizing* some resource signals, we are unaware of any work to use them to *predict* resource signals.

A signal may be predictable not only from its own history, but also from the history of other signals in the system. We plan to explore how such cointegration can benefit prediction. It may be possible to drastically reduce the number of measurements needed to maintain a particular level of predictability. For example, Vetter has shown that dynamic statistical projection pursuit can be used to determine the small subset of resource signals that are interesting from a performance visualization point of view [133]. Perhaps this approach, or even simple principle components analysis [66, pp 76–80], can be used to predict one signal's behavior using another signal.

Independent of any particular approach to prediction, it is a natural to ask how *inherently* predictable an arbitrary resource signal is. The effort that we as systems researchers should invest in attempting to predict it is in direct proportion to the answer. Unfortunately, while there are several theoretical frameworks for thinking about the question, there is currently no practical method for reducing a resource signal to a scalar predictability metric. In practice, a person with the appropriate skill set needs to read the tea leaves. We plan to continue to think about how to automate this process. It seems like it should be possible to create a useful heuristic procedure for resource signals.

RPS simplifies the final two steps of our resource signal methodology, but the remainder of the methodology requires considerable human involvement. In general, much of the process of characterizing signals and deciding on appropriate predictive models is highly heuristic, but those heuristics are not explicit, and therefore are difficult to automate. This should not be the case. We should be able to produce a system that automatically characterizes a new resource signal and then produces a predictor for it. Such a tool would fill a similar role to that of tools that characterize hardware features [144].

7.3.3 Resource scheduler models

A resource signal emerges when a workload is supplied to a scheduler that coordinates access to an underlying resource. A resource scheduler model, such as the running-time advisor in Figure 7.1, transforms resource signals (including predictions) into predictions of application-level performance metrics. The more accurately we can model the resource scheduler, the better the performance predictions are. We plan to improve the Unix scheduler model we developed in Chapter 5, and to develop new models for other resources, such as one that transforms from network bandwidth to the transfer time of messages. At this point, we are aware of no model of the Unix scheduler that can handle its priority scheme, for example.

We also need to encourage the developers of resource schedulers such as the Unix scheduler to introduce

more hooks for measuring their behavior—we need more resource signals, and we need them at a finer granularity. For example, the Digital Unix scheduler is invoked 1024 times per second by default, but the load averages are limited to 2 second timescales. The default situation on other operating systems is even worse.

7.3.4 Adaptation advisors

A real-time scheduling advisor offers one form of adaptation advice (a host recommendation) to achieve one kind of application goal (a deadline). This is a form of application-level scheduling [16], and many other such adaptation advisors are possible. The layered structure of Figure 7.1 means that additional kinds of services can share underlying layers. Other scheduling or adaptation advisors—load balancers, for example—can make use of the running time advisor. The running time advisor offers confidence intervals, such as those used in stochastic scheduling [117] of parallel applications. We plan to explore other kinds of adaptation advisors, although we have no immediate plans other than to improve the real-time scheduling advisor by incorporating network bandwidth predictions and an advisor for the transfer time of messages.

7.3.5 Application workloads

Few distributed interactive applications currently exist, in large part because developers are unwilling to commit themselves before being convinced that the furies of distributed computing environments can be tamed. The resulting dearth of application workloads makes the design of adaptation mechanisms, policies, and systems very difficult. The situation can be improved in two ways. First, we can better characterize interactive applications, such as Dv visualizations, that are necessarily already implemented in distributed form. The research community would benefit considerably from the widespread availability of such characterizations, or, even better, traces of their execution showing resource demands. Second, we can instrument existing non-distributed applications with the goal of presaging the workloads of their future distributed versions. Instrumenting commonly used, high level packages such as Vtk [118] seems to offer the most potential here. Another option is to instrument interactive applications that are based on scripting, such as GIMP, GNU's image editor program.

Appendix A

Application-oriented Prediction

Although this thesis argues for basing real-time scheduling advisors on explicit resource-oriented prediction using statistical signal processing, we did not discard application-oriented prediction out of hand. In fact, we began our research using the application-oriented approach. This appendix briefly reports on this work. The application-oriented approach can actually be made to perform quite well in limited situations if we focus solely on maximizing the fraction of deadlines met without concern for scalability, sharing of information among applications, or extensibility to other kinds of advisors. In contrast, the resource-oriented approach provides both high performance from the point of view of the application and the other benefits described in Chapter 1.

The design space of the application-oriented approach is vast. We studied a simple variant in which the advisor essentially uses a history of previous successes and failures or a history of running times and deadlines to predict which host will most likely meet the deadline at the current time. Unlike the resource-oriented advisor described in Chapter 6, the interface of the application-oriented advisor requires feedback from the application. This feedback is recorded in a local history from which predictions are made. Because this history resides within a specific advisor (indeed, within a specific process of the application), it is unavailable to other advisors. This is also in strong contrast to the resource-oriented advisor, where each resource maintains its own history, which is available to all the advisors in the system.

In this appendix, we describe the interface for this application-oriented approach and nine different algorithms that implement it. We then evaluate these algorithms using a simulation that is based on the load traces that we describe in Chapter 3. The evaluation focuses on the fraction (percent) of deadlines met as the sole metric of performance. In Chapter 6, we introduce other appropriate metrics. It is important to note that this evaluation is not fully randomized and does not approach the scale and depth of the evaluations of the different stages of the resource-oriented real-time scheduling advisor in Chapters 4–6.

The point of this chapter is simply to touch on the possibilities of the application-oriented approach. The most significant result is that one of the algorithms we developed is both tractable and near-optimal under most conditions in the limited situations we studied. There are several other lessons to draw from our results: (1) There is clearly significant opportunity for an application-oriented or resource-oriented prediction-based real-time scheduling advisor to exploit; (2) An application-oriented real-time scheduling advisor can make a significant difference as to whether a task meets its deadline or not; (3) Simple, statistical algorithms, even with randomization, are insufficient, behave inconsistently, and are inappropriate for the application-oriented approach; (3) Complex algorithms such as neural networks work reasonably well but are clearly too expensive to use in practice. Our near optimal algorithm is simple enough to be practical, but is not so simple that it is useless.

Name	Complexity		Timingsμ-sec		
	Time	Space	(4,50,0.1)	(8,50,0.1)	(16,50,0.1)
Optimal(RC)	-	-	-	-	-
Constant	$O(1)$	$O(1)$	0.244	0.244	0.244
Random	$O(1)$	$O(1)$	0.498	0.537	0.546
Mean	$O(N)$	$O(N)$	0.957	1.532	2.710
WinMean(W)	$O(N)$	$O(NW)$	3.318	4.460	7.005
WinVar(W)	$O(NW)$	$O(NW)$	7.039	9.580	14.73
RandWinMean(W,P)	$O(NW)$	$O(NW)$	7.223	13.166	24.534
RandWinVar(W,P)	$O(NW)$	$O(NW)$	17.408	33.58	65.866
Confidence(W)	$O(NW)$	$O(NW)$	13.674	21.360	36.775
RandConfidence(W,P)	$O(NW)$	$O(NW)$	26.951	52.869	102.423
RangeCounter(W)	$O(N) + O(W)$	$O(NW)$	8.500	9.551	11.960
NeuralNet(W)	$O(N^2W^2)$	$O(N^2W^2)$	41807.630	175543.700	773379.700

Table A.1: Algorithm complexity and timings. The numbers (N, W, P) refer to the number of hosts, the window size, and the randomization probability. Timings were collected on a DEC 3000/400.

A.1   Real-time scheduling advisor interface

The interface to the application-oriented real-time scheduling advisor consists of two functions. The first is used to ask the advisor to select a host for the task. The second is used to inform the advisor of the results of that selection. The interface is as follows:

```
Host  RTAdviseTask(Host  hosts[],
                   double  tnom,
                   double  slack);

int   RTInformTask(Host    host,
                   double  tnom,
                   double  slack,
                   double  tact);
```

The application uses `RTAdviseTask` to pose the following scheduling problem: Choose a host from hosts such that a task with nominal running time tnom (t_{nom}), if started now, will complete in $(1 + slack)t_{nom}$ seconds or less. The real-time scheduling advisor responds with the selected host. The application runs the task on the selected host and then reports the results to the advisor through the `RTInformTask` call. The application specifies the host on which the task ran, its nominal time, the slack, and the actual running time, `tact` (t_{act}).

A.2   Implementing the interface

The interface described in the previous section obviously can be provided by many different algorithms. We developed nine such algorithms which operate on the history of task executions provided by the `RTInformTask` call. Most of the algorithms are the simple statistical algorithms that one is led to when initially thinking about the problem. A common failing of these is that they fall into modes where they refuse to "explore" the space of possible task mappings and so can get "stuck" giving unfortunate mappings repeatedly. We added randomization to these initial algorithms to attempt to avoid this. The two remaining algorithms include a neural network algorithm and the *RangeCounter(W)* algorithm, which is nearly optimal most of the time. This section describes each of these algorithms. Table A.1 shows the asymptotic

complexity and actual running times of all the algorithms. It also includes three other algorithms we use for comparison purposes, *Optimal(RC)*, which as described in Section A.3, *Random*, and *Constant*. *Constant* simply always picks the same host while *Random* always picks a host at random.

The simplest algorithms include *Mean*, *WinMean(W)*, and *WinVar(W)*. For each host *Mean* computes the mean running time over the entire history of executions on that host and chooses the host with the lowest mean. *WinMean(W)* is similar, except the mean is computed over the last W executions on the host. *WinVar(W)* chooses the host with the lowest variance in running time over the last W executions. *RandWinMean(W,P)* and *RandWinVar(W,P)* select a random host with probability P and act like their non-randomized counterparts with probability $1 - P$.

Confidence(W) assumes that running times are picked from a normal distribution.[1] It computes the sample mean and variance over the last W executions on the host and parameterizes a normal distribution with these values. The host whose normal has the highest cumulative probability over the $[-\infty, (1 + slack)t_{nom}]$ range is chosen. The *RandConfidence(W,P)* algorithm chooses a random host with probability P and performs like *Confidence(W)* with probability $1 - P$.

The *RangeCounter(W)* algorithm can be thought of as a non-linearly filtered version of the *Confidence(W)* algorithm. Each host is assigned a quality value, initially zero. To choose a host, the following steps are taken: If no host has a quality value significantly higher than zero, a random host is chosen, otherwise the host with the highest quality value is chosen. Next, the quality values of all hosts are reduced (aged) by a small amount. If the choice of host is successful, then the chosen host's quality value is increased by the inverse of our statistical confidence in it. As with the *Confidence(W)* algorithm, we compute the sample mean and variance of the past W executions, parameterize a normal distribution with them, and compute the cumulative probability of that distribution over $[-\infty, (1 + slack)t_{nom}]$. We increase the host's quality value by the *inverse* of the cumulative probability. If the choice of host is unsuccessful, we divide the chosen host's quality value by two. *RangeCounter(W)* rewards good choices based on how unlikely they appear to be able to bear fruit and punishes bad choices equally. The idea is to encourage exploration and risk taking, but to react quickly and decisively to failure. The aging process discourages complacency.

The *NeuralNet(W)* algorithm consists of a multilayer feed-forward neural network [90] where each host contributes W inputs, the durations of the last W executions on that host. For N hosts, the intermediate layer has $NW/2$ nodes and the output layer has N nodes. The network is trained to drive the output representing the best host to the highest output value. To choose a host, the inputs are propagated forward through the network and the host with the highest corresponding output value is chosen. If the choice is successful, we reinforce the chosen output to unity and the remaining outputs to their present values and propagate these outputs back through the network. Similarly, if the choice is unsuccessful, we reinforce the chosen output to zero and the remaining outputs to their present values and propagate back. Essentially, Each choice/response pair is an iteration of the BACKPROP neural network training algorithm. We continuously train the network in operation.

A.3 Simulator and methodology

To develop and evaluate our algorithms, we built a simulator that allows us to simulate executing a task in a loop:

```
for (i=0;i<N;i++) {
    GetTask(tnom,slack);
    host=RTAdviseTask(hosts, tnom, slack);
    tnow=GetCurrentTime();
```

---

[1]A variety of other distributions were also examined.

```
    tact=RunTask(host, tnow, tnom);
    RTInformTask(host, tnom, slack, tact);
}
```

where the nominal running time `tnom` (t_{nom}) of the task is either constant or chosen from an enumerated distribution. The nominal running time is the time the task would take to execute under zero-load conditions on the slowest host. The task begins executing at `tnow` (t_{now}), which is determined just before the task is run on the selected host. For the duration of the loop, the algorithm is the same, so time advances according to the scheduling decisions made by the algorithm under test, just as it would in the real world.

Given the load trace of the host the algorithm has chosen, t_{now}, and t_{nom}, we use a simple model to simulate the actual running time, t_{act}, of the task on that host at that time. The running time is computed by the `RunTask` function, which bases it on a load trace. Consider a functional form of the load trace $\langle l_i \rangle$ so that $load(t) = l_i$ for $i\Delta \le t < (i + 1)\Delta$, where Δ is the sampling interval of the load trace. The actual running time, t_{act} is the value which satisfies

$$\int_{t_{now}}^{t_{now}+t_{act}} \frac{\beta}{1 + load(t)} dt = t_{nom}$$

where β is the raw speedup of the host over the slowest host. We have found that the compute time estimates given by this model correlate strongly (> 0.99 coefficient of correlation) with the actual computation time.

In addition to being run according to the advise of the algorithm under test, the task is also run (in simulation) according to a randomly chosen host, and according to the optimal choice. The details of each of these runs are written to a file.

A.3.1 Optimal algorithm

Using the model for computing running times from load traces, the simulator not only executes the task on the host the algorithm chooses, but on every host. This lets the simulator compute what an optimal algorithm would do if invoked at this point: it picks a host whose running time is less than the deadline, if one is available. It is important to note that simulated time evolves according to the algorithm under test, thus the measured performance of the optimal algorithm depends slightly on the algorithm under test. In practice, this varies only marginally across the algorithms we test. We present the performance of the optimal algorithm with respect to our near optimal algorithm, *RangeCounter(W)*. In this incarnation, the optimal algorithm is referred to as *Optimal(RC)*.

A.4   Scenarios and methodology

We constructed six different configurations of hosts (or *scenarios*) from the load traces discussed in Chapter 3. The scenarios consist of traces of hosts with different mean loads and different mean epoch lengths. In Figure A.1 we plot each load trace according to its mean load and its mean epoch length. The six scenarios we discuss here are then highlighted, labeled, and tabulated. We categorize the load of a scenario by whether it contains only machines with low (≈ 0.2) mean loads, only machines with high mean loads (≈ 1.1), or a mixed combination of machines in the former categories. Similarly, we categorize the epoch length of a scenario by whether all of its machines have short (150-300 seconds) epochs, long (400-500 seconds) epochs, or a combination of the former. The labels consist of a number and two letters. The number represents the number of hosts in the scenario, the first letter represents whether the load category is (S)mall, (M)ixed, or (L)arge, and the second letter indicates whether the epoch category is (S)hort, (M)ixed, or (L)ong. Notice that there are nine combinations, but our data limits us to exploring only six of the combinations.
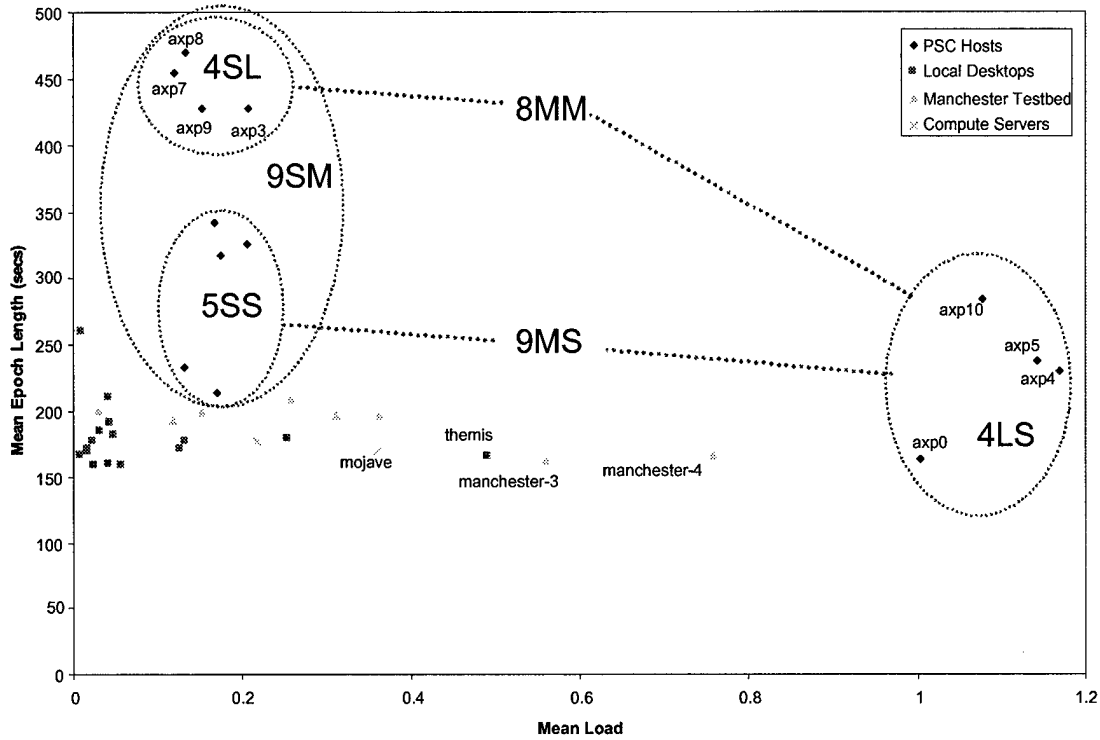
	Load Category		
Epoch Category	Small	Mixed	Large
Short	5SS	9MS	4LS
Mixed	9SM	8MM	
Long	4SL		

Figure A.1: Application prediction scenarios.

The scenarios presented here contain only the PSC hosts. We have also done limited simulations using other scenarios involving the remaining machines, but there is no space to present them here. The advantage of using the PSC hosts is that they nicely cover most of the space formed by the load and epoch categories.

For each of the six scenarios, we looked at eight different values of the nominal running time t_{nom}, and twelve different values of the upper timing bound $slack$, ranging from 0 to 2. This forms 576 different cases and we simulated 100,000 consecutive tasks for each case, for a total of 57.6 million simulated tasks. In addition, for each scenario, we simulated what happens when t_{nom} is picked randomly from a set of four values (25,50,75,100 ms) on each iteration, and $slack$ is varied as before. This forms another 72 cases and 7.2 million tasks.

A.5 Evaluation results

In this section, we present three slices through our simulation results to make presentation tractable. The first two slices show how the algorithms' performance varies with nominal running time and with slack. The third slice shows how the performance varies when the nominal time is permitted to vary, as if different paths are taken within the task. The performance of an algorithm is the percentage of tasks that meet their
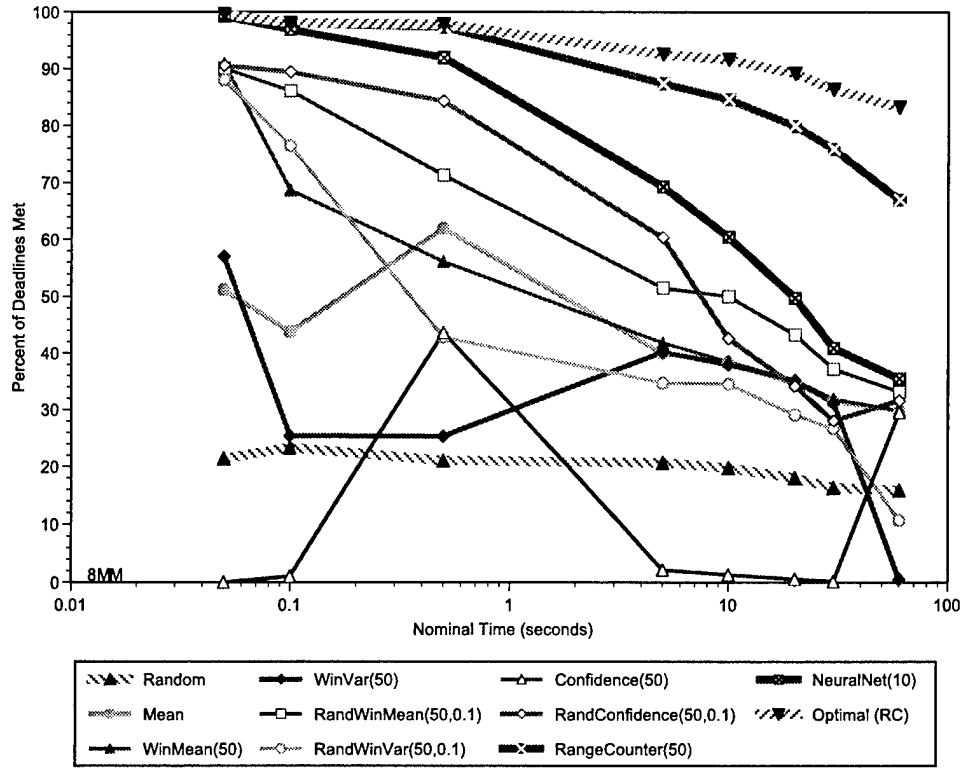
Figure A.2: The effect of varying nominal running time for the 8MM scenario at a slack of zero.

deadlines $((1 + slack)t_{nom})$.

Our discussion covers all of the simulations. However, in order to simplify our presentation, we show graphs for only two representative scenarios, 8MM and 4LS. As can be seen in Figure A.1, 8MM contains eight machines with mixed load and epoch lengths, while 4LS contains four machines with high loads and short epoch lengths. To show the effect of nominal running time, only 8MM is shown since 4LS does not provide any additional insight. The other scenarios show similar results.

For the algorithms that have parameters, namely W, the window of previous running times recorded for a host, and P, the probability of a random assignment (for the randomized algorithms), we have fixed the parameters to $W = 50$ (10 for *NeuralNet(W)*) and $P = 0.1$. The performance of the algorithms given these values is representative of how they perform with the other, non-extreme, values we studied. The relative performance of the algorithms is not very sensitive to the choice of these values.

A.5.1   Performance versus nominal time

Figure A.2 shows how the different algorithms perform when scheduling tasks with different nominal running times in a representative scenario, 8MM. The other scenarios yield similar results. The nominal times range from 50 milliseconds to 60 seconds. The x-axis is the nominal time, t_{nom}, ranging from 50 milliseconds to 60 seconds, and the y-axis is the percentage of tasks that meet their deadlines. The slack is set to zero.

Notice that there is a substantial gap between the performance of the optimal algorithm (*Optimal(RC)*) and the performance of random assignment (*Random*). Although we have elided the results for clarity, it is also never the case that always choosing the same host works well. We find similar results for the other five scenarios, with differences ranging from 40 to 80 percent. This suggests that a good algorithm can make a
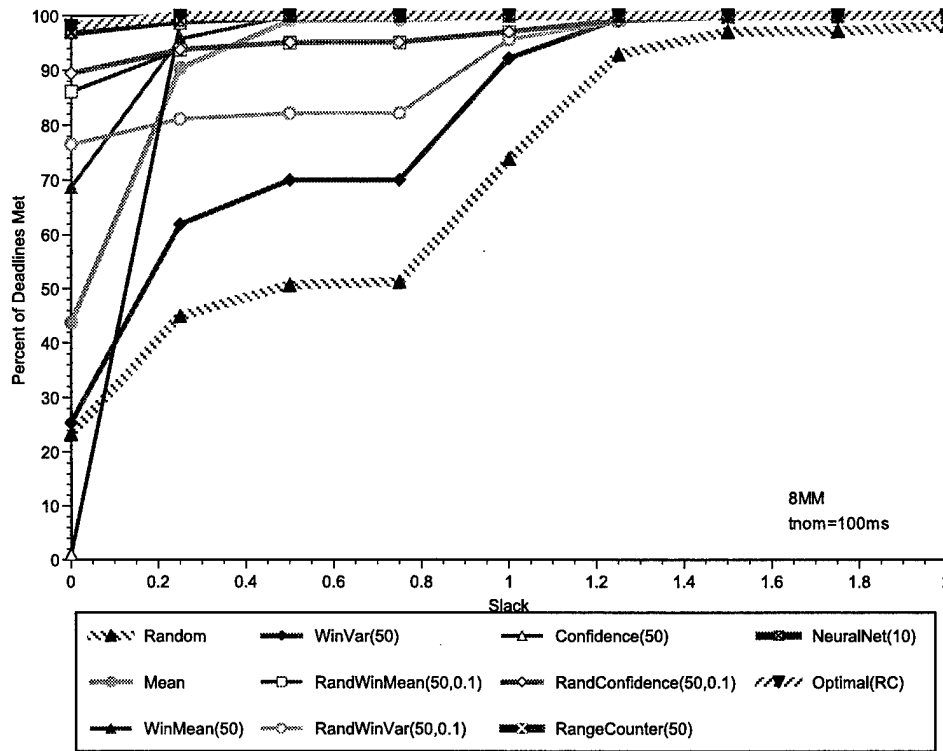
Figure A.3: The effect of slack for the 8MM scenario. $t_{nom} = 100ms$ and the slack varies from 0 to 2.

large difference.

Interestingly, the simple statistics-based algorithms and their randomized variants perform rather badly for the most part, some worse than even *Random*. Further, they tend to behave inconsistently. For example, *WinVar(W)* is about 35% better than *Random* at 50 milliseconds, then abruptly declines with increasing t_{nom} only to improve later and finally become completely ineffectual at 60 seconds. In another scenario, we find it near optimal for some nominal times and yet collapsing to worse than *Random* for nearby times. On the other hand, *NeuralNet(W)* and *RangeCounter(W)* behave consistently well and near-optimally for short nominal times less than one second. In fact, *RangeCounter(W)* is the best algorithm for all scenarios and nominal times except for the 60 second time in the 4LS scenario.

A.5.2 Performance versus slack

The effect of loosening the slack on the performance of the algorithms is quite interesting. In Figure A.3 we show the effect in the 8MM scenario and in Figure A.4 we show the effect in the 4LS scenario. The x-axis in each graph is the slack value while the y-axis is the percentage of tasks that meet the deadline. The nominal running time is fixed at 100 milliseconds and the slack varies from 0 to 2. The remaining four scenarios show effects similar to the 8MM scenario (Figure A.3). Other choices of nominal running time lead to similar graphs in each of the scenarios. This is different from the behavior we observed in our evaluation of the resource-oriented real-time scheduling advisor (Chapter 6). The difference is due to the fact that the simulation approach that we use here does not take into account the priority boost we describe in Chapter 5.

In Figure A.3, notice that there is great differentiation between the optimal algorithm and random assignment as the slack gets tighter. The same holds true in Figure A.4 for slacks greater than one. For this
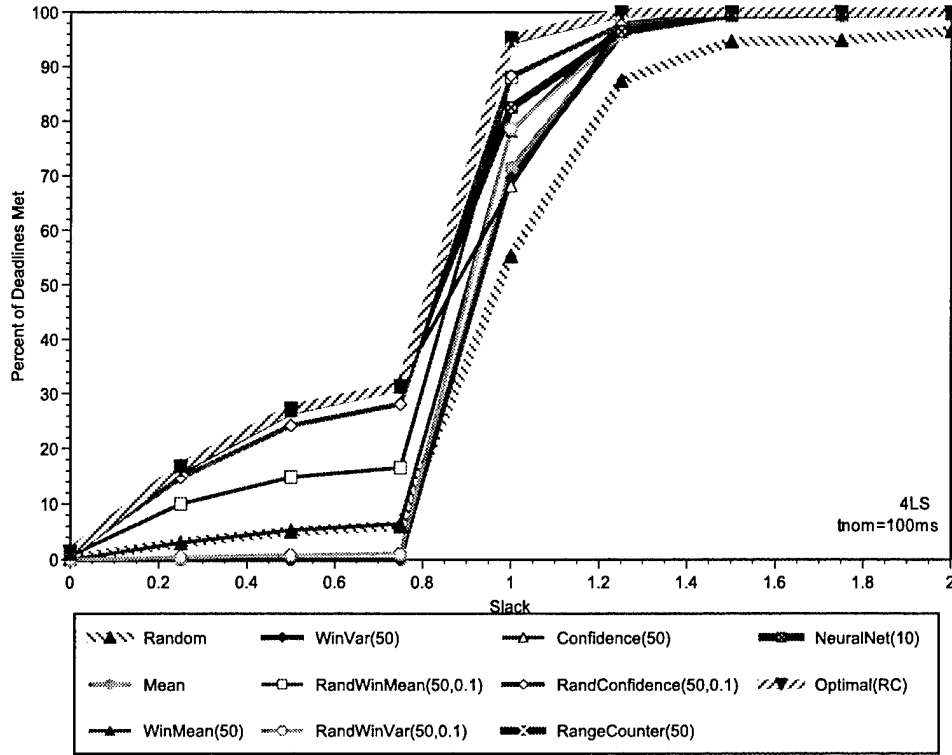
Figure A.4: The effect of slack for the 4LS scenario. $t_{nom} = 100ms$ and the slack varies from 0 to 2.

scenario, there are so few machines that can meet the tighter deadlines that the gap between *Optimal(RC)* and *Random* declines severely with slack less than one.

In both figures, the simpler algorithms require a reasonably large amount of additional slack before their performance catches up to *RangeCounter(W)* and *NeuralNet(W)*. This is also the case for the other scenarios, although the difference tends to be slightly less for scenarios with only unloaded machines. In Figure A.3, the simpler algorithms require between 0.25 and and 1.0 of extra slack to get to within an absolute 10% of the performance of *RangeCounter(W)* and *NeuralNet(W)*. This corresponds to deadlines on the order of twice the running time of the task, which is a significant amount of extra time. For Figure A.4, we see a great upsurge in performance as the slack is relaxed to around 1.0 (ie, deadlines that are double the nominal running time). Before this point, there are very few opportunities in which it is possible to meet the deadlines. However, *RangeCounter(W)* manages to ferret out these opportunities and manages to closely approximate the optimal algorithm's performance across the board. It behaves well even when choices are very constrained.

A.5.3   Performance with varying nominal times

Figures A.5 (8MM scenario) and A.6 (4LS scenario) show the effect of loosening the deadline (t_{max}) when the nominal running time is picked randomly from a set of values. In each graph, the nominal time is randomly picked from 25, 50, 75, and 100 milliseconds for each call. Intuitively, this can be seen as following different control paths through the task. The x-axis is the deadline and the y-axis is the performance. We sweep the deadline from 25 ms to 300 ms. The linear upward trend on the first part of each graph is due to the varying nominal time. At 50 milliseconds, only about 50% of the calls have a nominal time of 50 milliseconds or less, so even the optimal algorithm can only manage to map 50% of the calls successfully,

Figure A.5: The effect of different timing constraints for the 8MM scenario. $t_{nom} = 100ms$ and the constraint is $[0, t_{max}]$ where t_{max} varies from 100ms to 300ms.
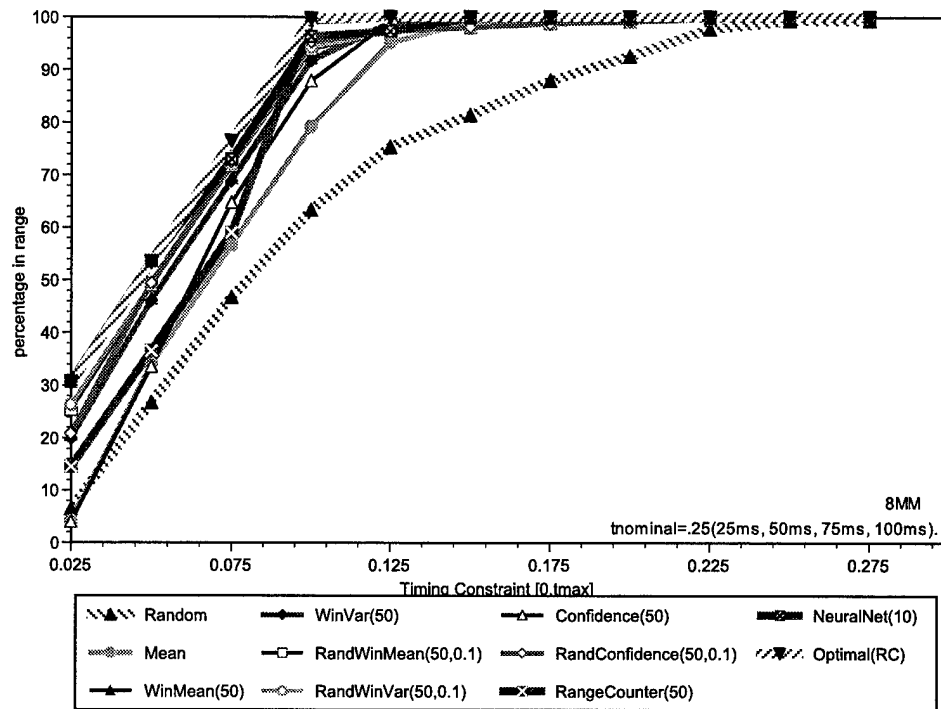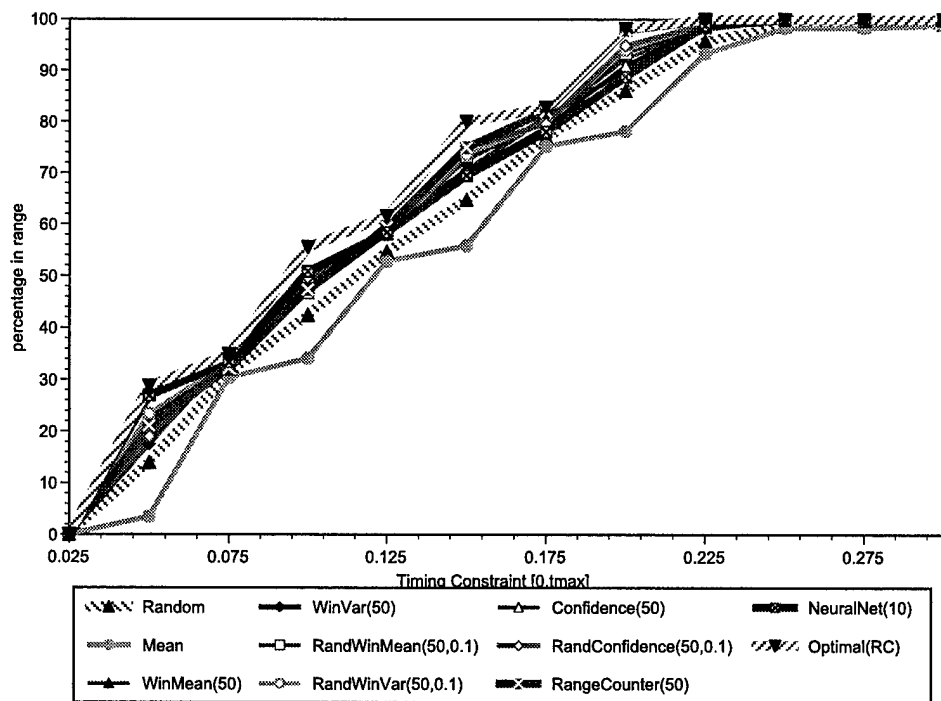

Figure A.6: The effect of different timing constraints for the 4LS scenario. $t_{nom} = 100ms$ and the constraint is $[0, t_{max}]$ where t_{max} varies from 100ms to 300ms.

even given totally idle machines.

Notice the separation between the optimal algorithm and random mapping, suggesting opportunity for a mapping algorithm. The separation is smaller than in the previous cases, since at any particular deadline except for 25 ms, there are some number of calls that have significant slack. For example, at 50 ms, 25% of the calls have a nominal time of 25 ms and thus have a *slack* of 1. This also benefits the simple algorithms and decreases the spread in performance. However, the spread can still be significant. For example, in the 8MM scenario (Figure A.5), the spread is as high as an absolute 40% of the calls.

An important point here is that *RangeCounter(W)*, although it does well, behaves somewhat erratically in the face of the varying nominal times and is not clearly superior to the other algorithms until the deadline has been relaxed to encompass the longest of the different nominal times. That is, when some varying nominal times exceed the deadline, *RangeCounter(W)* can behave badly. In some cases, it is as much as 20% down from the optimal algorithm and *NeuralNet(W)*. This presents opportunity for improvement.

A.5.4 Conclusions

There are several conclusions to be drawn from our results. The first conclusion is that there exists significant opportunity for real-time scheduling advisor. By choosing the right host, an advisor can greatly increase the chance that a task will meet its deadline. This is true of both application-oriented and resource-oriented advisors. The second conclusion is that, for advisors that rely on application-oriented prediction, it is the case that simple summary statistic-based algorithms are insufficient. In some cases, simple algorithms such as *Mean*, *WinMean(W)*, *WinVar(W)*, and *Confidence(W)* can actually perform worse than *Random*. Worse, they can behave erratically in the face of different nominal running times. Adding randomization to these algorithms smoothes them and improves their overall performance, but it still does not make them competitive with the best algorithms, *NeuralNet(W)* and *RangeCounter(W)*.

The load trace results we presented in Chapter 3 can shed some light on why the simple statistical algorithms perform so badly. Recall that the load traces exhibit both short- and long-range autocorrelation. Clearly, consecutive load values are *not* chosen independently according to some stationary probability distribution function. However, this is precisely the underlying assumption made by the simple statistical algorithms! Further, even if, in the long term, some pdf does emerge, it is highly unlikely that this pdf is applicable over the short term, given the epochal behavior discussed in Section 3.6. Of course, we might expect that with a sufficiently long nominal time, the simple statistical algorithms might work better, but even there there is some doubt. Unfortunately, the histograms of the load traces (which we elide for space reasons) do not appear to be fit any analytic distribution we are aware of.

NeuralNet(W) and *RangeCounter(W)* appear to be better able to deal with the properties exhibited by the load traces because they behave non-linearly. This is probably a great benefit when transitioning from one epoch to the next because it suggests that the internal state of the algorithms will change drastically at these points, which is, of course, what we desire. Another interesting point is that as the nominal running time grows, there are fewer and fewer training examples per epoch. This offers a possible explanation for why *NeuralNet(W)*, which behaves near optimally for short nominal times, quickly declines as the nominal times grow. Often, as neural networks are trained, they tend to converge only slowly to a good set of weights. After some large number of training sets, there is an abrupt convergence, beyond which any additional training tends to make the neural network slowly worse. We suspect that what we are seeing is that with a long enough nominal time there is simply an insufficient number of training samples in an epoch for *NeuralNet(W)* to make that abrupt convergence.

RangeCounter(W) exhibits the best performance in nearly every case, and behaves relatively smoothly. For nominal times less than one second, it behaves almost optimally, while its performance degrades gradually for longer nominal times. Because it is near optimal even with the tightest slack, it is no surprise that

it remains so as we relax the slack. Further, while *NeuralNet(W)* is also near optimal under conditions of tight slack and short nominal times, none of the other tractable algorithms are especially close, and some require considerable loosening of the slack before they become anywhere near optimal. The failings of *RangeCounter(W)* are that it behaves suboptimally in the presence of very long nominal times. In some sense, *RangeCounter(W)* tends to become specialized too quickly, which is the reverse of *NeuralNet(W)*'s problem. *NeuralNet(W)* seems to remain sufficiently general to do well with varying nominal times.

Bibliography

[1] Henry Abarbanel. *Analysis of Observed Chaotic Data*. Institute for Nonlinear Science. Springer, 1996.

[2] Adobe Corporation. *Adobe Photoshop 4.0 User's Guide*, 1997.

[3] M. Aeschlimann, P. Dinda, L. Kallivokas, J. Lopez, B. Lowekamp, and D. O'Hallaron. Preliminary report on the design of a framework for distributed visualization. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 1833–1839, Las Vegas, NV, June 1999. CSREA Press.

[4] Jont B. Allen and David A. Berkley. Method for simulating small-room acoustics. *Journal of the Acoustical Society of America*, 65(4):943–950, April 1979.

[5] Amaranth Project. Amaranth: Probabilistically guaranteed quality of service for distributed computing systems. (web page). http://www.cs.cmu.edu/afs/cs/project/ices-amaranth/www/.

[6] J. Arabe, A. Beguelin, B. Lowekamp, M. Starkey E. Seligman, and P. Stephan. Dome: Parallel programming in a heterogeneous multi-user environment. Technical Report CMU-CS-95-137, Carnegie Mellon University, School of Computer Science, April 1995.

[7] Olaf Arndt, Bernd Freisleben, Thilo Kielmann, and Frank Thilo. Dynamic load distribution with the winner system. In *Proceedings of Workshop Anwendungsbezogene Lastverteilung (ALV'98)*, pages 77–88, March 1998. Also available as Technische Universitt Mnchen Technical Report TUM-I9806.

[8] Hari Balakrishnan, Mark Stemm, Srinivasan Seshan, and Randy H. Katz. Analyzing stability in wide-area network performance. In *Proceedings of SIGMETRICS'97*, pages 2–12. ACM, 1997.

[9] Hesheng Bao, Jacobo Bielak, Omar Ghattas, Loukas F. Kallivokas, David R. O'Hallaron, Jonathan R. Shewchuk, and Jifeng Xu. Large-scale Simulation of Elastic Wave Propagation in Heterogeneous Media on Parallel Computers. *Computer Methods in Applied Mechanics and Engineering*, 152(1–2):85–102, January 1998.

[10] M. R. Barbacci and J. M. Wing. DURRA: A task-level description language. In *Proceedings of The International Conference on Parallel Processing*, pages 370–376, August 1987.

[11] James B. Bassingthwaighte, Daniel A. Beard, Donald B. Percival, and Gary M. Raymond. Fractal structures and processes. In Donald E. Herbert, editor, *Chaos and the Changing Nature of Science and Medicine: An Introduction*, number 376 in AIP Conference Proceedings, pages 54–79. American Institute of Physics, April 1995.

195

[12] Sabyasachi Basu, Amarnath Mukherjee, and Steve Klivansky. Time series models for internet traffic. Technical Report GIT-CC-95-27, College of Computing, Georgia Institute of Technology, February 1995.

[13] BBN Corporation. Distributed spatial technology laboratories: Openmap. (web page). http://javamap.bbn.com/.

[14] Durand R. Begault. *3-D Sound For Virtual Reality and Multimedia*. AP Professional, 1994.

[15] Jan Beran. Statistical methods for data with long-range dependence. *Statistical Science*, 7(4):404–427, 1992.

[16] Francine Berman and Richard Wolski. Scheduling from the perspective of the application. In *Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing HPDC96*, pages 100–111, August 1996.

[17] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[18] Azer Bestavros. Load profiling: A methodology for scheduling real-time tasks in a distributed system. In *Proceedings of ICDCS '97*, May 1997.

[19] Azer Bestavros and Dimitrios Spartiotis. Probabilistic job scheduling for distributed real-time applications. In *Proceedings of the First IEEE Workshop on Real-Time Applications*, May 1993.

[20] Andrew D. Birrel and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[21] Avrim Blum and Carl Burch. On-line learning and the metrical task system problem. In *Proceedings of the 10th Annual Conference on Computational Learning Theory (COLT '97)*, pages 45–53, 1997.

[22] J. Bolliger, T. Gross, and U. Hengartner. Bandwidth modelling for network-aware applications. In *Proceedings of Infocomm'99*, pages 1300–1309, 1999.

[23] George E. P. Box, Gwilym M. Jenkins, and Gregory Reinsel. *Time Series Analysis: Forecasting and Control*. Prentice Hall, 3rd edition, 1994.

[24] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation protocol (RSVP) – version 1 functional specification. Internet RFC 2205, September 1997. ftp://ftp.isi.edu/in-notes/rfc2205.txt.

[25] Peter J. Brockwell and Richard A. Davis. *Introduction to Time Series and Forecasting*. Springer-Verlag, 1996.

[26] Nat Brown and Charlie Kindel. Distributed component object model protocol – dcom/1.0. Technical report, Microsoft, May 1996. http://ds1.internic.net/internet-drafts/draft-brown-dcom-v1-spec-00.txt.

[27] Kristian Paul Bubendorfer. Resource based policies for load distribution. Master's thesis, Victoria University of Wellington, August 1996.

[28] R. Chin and S. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[29] P. Dinda, B. Lowekamp, L. Kallivokas, and D. O'Hallaron. The case for prediction-based best-effort real-time systems. In *Proc. of the 7th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 1999)*, volume 1586 of *Lecture Notes in Computer Science*, pages 309–318. Springer-Verlag, San Juan, PR, 1999. Extended version as CMU Technical Report CMU-CS-TR-98-174.

[30] Peter A. Dinda. The statistical properties of host load. *Scientific Programming*, 7(3,4), 1999. A version of this paper is also available as CMU Technical Report CMU-CS-TR-98-175. A much earlier version appears in LCR '98 and as CMU-CS-TR-98-143.

[31] Peter A. Dinda and David R. O'Hallaron. An evaluation of linear models for host load prediction. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC '99)*, pages 87–96, August 1999. Extended version available as CMU Technical Report CMU-CS-TR-98-148.

[32] Peter A. Dinda and David R. O'Hallaron. An extensible toolkit for resource prediction in distributed systems. Technical Report CMU-CS-99-138, School of Computer Science, Carnegie Mellon University, July 1999.

[33] Peter A. Dinda and David R. O'Hallaron. Realistic CPU workloads through host load trace playback. In *Proc. of 5th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR2000)*, May 2000. To appear.

[34] Lisa Cingiser DiPippo, Roman Ginis, Michael Squadrito, Steven Wohlever, Victor Fay Wolfe, Igor Zykh, and Russel Johnston. Expressing and enforcing timing constraints in a CORBA environment. Technical Report 97-252, Department of Computer Science, University of Rhode Island, February 1997.

[35] DIS Steering Committee. The dis vision: A map to the future of distributed simulation. Technical Report Version 1, DIS Steering Committee, May 1994.

[36] Terrence L. Disz, Michael E. Papka, Michael Pellegrino, and Rick Stevens. Sharing visualization experiences among remote virtual environments. In *Proceedings of the International Workshop on High Performance Computing for Computer Graphics and Visualization*, pages 217–237, 1995.

[37] S. Dwarkadas, A. Cox, and W. Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proc. Sixth Intl. Conf. on Architectural Support for Prog. Languages and Operating Systems (ASPLOS VI)*, Boston, Oct 1996. ACM.

[38] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.

[39] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation*, 6:53–68, March 1986.

[40] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. The limited performance benefits of migrating active processes for load sharing. In *SIGMETRICS '88*, pages 63–72, May 1988.

[41] D. W. Embley and G. Nagy. Behavioral aspects of text editors. *ACM Computing Surveys*, 13(1):33–70, January 1981.

[42] Yasuhiro Endo, Zheng Wang, J. Bradley Chen, and Margo Seltzer. Using latency to evaluate interactive system performance. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, 1996.

[43] Anja Feldman, Anna C. Gilbert, and Walter Willinger. Data networks as cascades: Investigating the multifractal nature of internet WAN traffic. In *Proceedings of ACM SIGCOMM '98*, pages 25–38, 1998.

[44] D. Ferrari. Client requirements for real-time communication services. *IEEE Communications Magazine*, 11(11):65–72, November 1990.

[45] D. Ferrari, A. Banerjea, and H. Zhang. Network support for multimedia - a discussion of the tenet approach. *Computer Networks and ISDN Systems*, 26(10):1167–1180, July 1994.

[46] Silvia M. Figueira and Francine Berman. Predicting slowdown for networked workstations. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC 97)*, pages 92–101, August 1997.

[47] Patricia Gomes Soares Florissi. *QosME: QoS Management Environment*. PhD thesis, Columbia University Computer Science Department, 1996.

[48] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[49] Chris Fraley. Fracdiff: Maximum likelihood estimation of the parameters of a fractionally differenced ARIMA(p, d, q) model. Computer Program, 1991. http://www.stat.cmu.edu/general/fracdiff.

[50] M. Garrett and W. Willinger. Analysis, modeling and genreation of self-similar VBR video traffic. In *Proceedings of SIGCOMM '94*, London, September 1994.

[51] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Mancheck, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, Massachusetts, 1994.

[52] G. A. Geist, James Arthur Kohl, and Phillip M. Papadopoulos. CUMULVS: Providing fault-tolerance, visualization, and steering of parallel applications. Technical report, Oak Ridge National Laboratory, 1996.

[53] D. K. Gifford. Weighted voting for replicated data. *ACM Transactions on Database Systems*, 16(4):150–159, 1979.

[54] Aniruddha Gokhale and Douglas C. Schmidt. Measuring the performance of communication middleware on high-speed networks. In *Proceedings of SIGCOMM 96*, August 1996. Extended version available as TR.

[55] C. W. J. Granger and Roselyne Joyeux. An introduction to long-memory time series models and fractional differencing. *Journal of Time Series Analysis*, 1(1):15–29, 1980.

[56] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, 1993.

[57] Nancy C. Groschwitz and George C. Polyzos. A time series model of long-term NSFNET backbone traffic. In *Proceedings of the IEEE International Conference on Communications (ICC'94)*, volume 3, pages 1400–4, May 1994.

[58] H. Haggen, H. Mueller, and G. M. Nielson, editors. *Focus On Scientific Visualization*. Springer-Verlag, 1993.

[59] Max Hailperin. *Load Balancing using Time Series Analysis For Soft Real-Time Systems with Statistically Periodic Loads*. PhD thesis, Stanford University, December 1993.

[60] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. In *Proceedings of ACM SIGMETRICS '96*, pages 13–24, May 1996.

[61] John Haslett and Adrian E. Raftery. Space-time modelling with long-memory dependence: Assessing ireland's wind power resource. *Applied Statistics*, 38:1–50, 1989.

[62] R. Hoare, H. G. Dietz, T. Mattox, and S. Kim. Bitwise aggregate networks. In *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing*, October 1996.

[63] J. R. M. Hosking. Fractional differencing. *Biometrika*, 68(1):165–176, 1981.

[64] Jyri Huopaniemi, Matti Karjalainen, Vesa Vaelimaeki, and Tommi Houtilainen. Virtual intstruments in virtual rooms — a real-time binaural room simulation environment for physical models of musical instruments. In *Proceedings of the International Computer Music Conference (ICMC '94)*, September 1994.

[65] H. E. Hurst. Long-term storage capacity of reservoirs. *Transactions of the American Society of Civil Engineers*, 116:770–808, 1951.

[66] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, Inc., 1991.

[67] E. Douglas Jensen. A real-time manifesto. http://www.realtime-os.com, 1996.

[68] E. Douglas Jensen, C Douglass Lock, and Hideyuki Tokuda. A time-driven scheduling model for real-time operating systems. In *Proceedings of the Real-Time Systems Symposium*, pages 112–122, Februrary 1985.

[69] Joseph A. Kaplan and Michael L. Nelson. A comparison of queueing, cluster, and distributed computing systems. Technical Report NASA TM 109025 (Revision 1), NASA Langley Research Center, June 1994.

[70] David A. Karr, David E. Bakken, John A. Zinky, and Thomas F. Lawrence. Towards quality of service for groupware. BBN Corporation (Submitted to ICDCS '99).

[71] G. Knittel. A parallel algorithm for scientific visualization. In *Proceedings of the 1996 International Conference on Parallel Processing (IPPS '96)*, volume 2, pages 116–123. IEEE, August 1996.

[72] A. Komatsubara. Psychological upper and lower limits of system response time and user's preferance on skill level. In G. Salvendy, M. J. Smith, and R. J. Koubek, editors, *Proceedings of the 7th International Conference on Human Computer Interaction (HCI International 97)*, volume 1, pages 829–832. IEE, August 1997.

[73] J. Kurose. Open issues and challenges in providing quality of service guarantees in high speed networks. *ACM Computer Communication Review*, 23(1):6–15, January 1993.

[74] James F. Kurose and Renu Chipalkatti. Load sharing in soft real-time distributed computer systems. *IEEE Transactions on Computers*, C-36(8):993–1000, August 1987.

[75] John Lehoczky. Real-time queueing theory. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, pages 186–195, 1996.

[76] John Lehoczky. Real-time queueing network theory. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, 1997.

[77] Will E. Leland and Teunis J. Ott. Load-balancing heuristics and process behavior. In *Proceedings of Performance and ACM SIGMETRICS*, volume 14, pages 54–69, 1986.

[78] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of ethernet traffic. In *Proceedings of ACM SIGCOMM '93*, September 1993.

[79] Christopher J. Lindblad, David J. Wetherall, and David L. Tennenhouse. The VuSystem: A programming system for visual processing of digital video. In *Proceedings of the ACM Multimedia 94*. ACM, October 1994.

[80] M. Litzkow, M. Livny, and M. W. Mutka. Condor — a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS '88)*, pages 104–111, June 1988.

[81] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[82] Bruce Lowekamp, Nancy Miller, Dean Sutherland, Thomas Gross, Peter Steenkiste, and Jaspal Subhlok. A resource monitoring system for network-aware applications. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 189–196. IEEE, July 1998.

[83] Bruce Lowekamp, David O'Hallaron, and Thomas Gross. Direct queries for discovering network resource properties in a distributed environment. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC99)*, pages 38–46, August 1999.

[84] MathSoft, Inc. *S-Plus User's Guide*, August 1997. See also http://www.mathsoft.com/splus.

[85] The Mathworks, Inc. *MATLAB System Identification Toolbox User's Guide*, 1996. see also http://www.mathworks.com/products/sysid.

[86] The Mathworks, Inc. *MATLAB User's Guide*, 1996. see also http://www.mathworks.com/products/matlab.

[87] Pankaj Mehra. *Automated Learning of Load-Balancing Strategies for a Distributed Computer System*. PhD thesis, University of Illinois at Urbana-Champaign, 1993.

[88] Pankaj Mehra and Benjamin W. Wah. Automated learning of workload measures for load balancing on a distributed system. In *Proceedings of IPPS '93*, pages III–263–III–270. CRC Press, August 1993.

[89] Nancy Miller and Peter Steenkiste. Network status information for network-aware applications. In *Proceedings of IEEE Infocom 2000*, March 2000. To Appear.

[90] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[91] Patrick R. Morin. The impact of self-similarity on network performance analysis. Technical Report Computer Science 95.495, Carleton University, December 1995.

[92] Matt W. Mutka and Miron Livny. The available capacity of a privately owned workstation environment. *Performance Evaluation*, 12(4):269–284, July 1991.

[93] Andy Myers, Peter Dinda, and Hui Zhang. Performance characteristics of mirror servers on the internet. In *Proceedings of the Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE INFOCOM '99)*, pages 304–312. IEEE, March 1999. (Volume I).

[94] Klara Nahrstedt and Jonathan M Smith. An application-driven approach to networked multimedia systems. In *Proceedings of the 18th Annual Conference on Local Area Computer Networks*, pages 361–368. IEEE, 1993.

[95] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997. To Appear.

[96] Brian D. Noble, M. Satyanarayanan, Giao T. Nguyen, and Randy H. Katz. Trace-based mobile network emulation. In *Proceedings of the ACM SIGCOMM Conference*, Cannes, France, September 1997.

[97] Object Management Group. Realtime corba: A white paper. http://www.omg.org, December 1996. In Progess.

[98] Object Management Group. The common object request broker: Architecture and specification (version 2.3.1). Technical report, Object Management Group, 1999.

[99] Katia Obraczka and Grig Gheorghiu. The performance of a service for network-aware applications. In *Proceedings of the ACM SIGMETRICS SPDT'98*, October 1997. (also available as USC CS Technical Report 97-660).

[100] Alan V. Oppenheim, Alan S. Willsky, and Ian T. Young. *Signals and Systems*. Prentice Hall, 1983.

[101] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley and Sons, Inc., New York, New York, 1996.

[102] C. Partridge, T. Mendez, and W. Milliken. Host anycasting service. IETF RFC 1546, November 1993.

[103] Vern Paxson and Sally Floyd. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[104] Andreas Polze, Gerhard Fohler, and Matthias Werner. Predictable network computing. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97)*, pages 423–431, May 1997.

[105] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in Fortran*. Cambridge University Press, 1986.

[106] A. Prochazka and Jan Uhlir, editors. *Signal Analysis and Prediction*. Applied and Numerical Harmonic Analysis Series. Springer Verlag, June 1998.

[107] Irfan Pyarali, Timothy H. Harrison, and Douglas C. Schmidt. Design and performance of an object-oriented framework for high-speed electronic medical imaging. *Usenix Computing Systems Journal*, 9(3), November/December 1996. earlier version appeared in the Usenix COOTS Conference, June, 1996.

[108] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shui Oikawa. Resource kernels: A resource-centric approach to real-time systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[109] Raj Rajkumar, Chen Lee, John Lehoczky, and Dan Siewiorek. A resource allocation model for QoS management. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1997.

[110] D. Rantzau, U. Lang, R. Lang, H. Nebel, A. Wierse, and R. Ruehle. Collaborative and interactive visualization in a distributed high performance software environment. In *Proceedings of the International Workshop on High Performance Computing for Computer Graphics and Visualization*, pages 207–216, 1995.

[111] S. Reinhardt, J. Larus, and D. Wood. Tempest and typhoon: User-level shared memory. In *Proceedings of ISCA '94*, pages 325 – 336. ACM, April 1994.

[112] V. J. Ribeiro, R. H. Riedi, M. S. Crouse, and R. G. Baraniuk. Simulation of non-gaussian long-range-dependent traffic using wavelets. In *Proceedings of ACM SIGMETRICS '99*, pages 1–12, May 1999.

[113] M. Rinard, D. Scales, and M. Lam. Jade: A high-level machine-independent language for parallel programming. *IEEE Computer*, 26(6):28–38, June 1993.

[114] Bikash Sabata, Saurav Chatterjee, Michael Davis, Jaroslaw J. Sydir, and Thomas F. Lawrence. Taxonomy for QoS specifications. In *Proceedings of the Third International Workshop on Object-Oriented Real-time Dependable Systems (WORDS '97)*, February 1997.

[115] Mehrdad Samadani and Erich Kalthofen. On distributed scheduling using load prediction from past information. Abstracts published in Proceedings of the 14th annual ACM Symposium on the Principles of Distributed Computing (PODC'95, pp. 261) and in the Third Workshop on Languages, Compilers and Run-time Systems for Scalable Computers (LCR'95, pp. 317–320), 1996.

[116] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[117] Jennifer M. Schopf and Francine Berman. Stochastic scheduling. In *Proceedings of Supercomputing '99*, 1999. Also available as Northwestern University Computer Science Department Technical Report CS-99-03.

[118] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-oriented Approach to 3D Graphics*. Prentice Hall, second edition edition, 1998.

[119] Wolfgang Schwabl, Johannes Reisinger, and Guenter Gruensteidl. A survey of MARS. Technical Report 16/89, Technische Universitaet Wien, October 1989.

[120] S. Seshan, M. Stemm, and R. H. Katz. SPAND: Shared passive network performance discovery. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and System (USITS)*, 97.

[121] C. E. Shannon. A mathematical theory of communication. *Bell System Tech. J.*, 27:379–423, 623–656, 1948.

[122] Niranjan G. Shivaratri and Phillip Krueger. Two adaptive location policies for global scheduling algorithms. In *Proceedings of ICDCS '90*, pages 502–509, May 1990.

[123] Jon Siegal. *CORBA Fundamentals and Programming*. John Wiley and Sons, Inc., 1996.

[124] Bruce Siegell and Peter Steenkiste. Automatic generation of parallel programs with dynamic load balancing. In *Proceedings of the Third International Symposium on High-Performance Distributed Computing*, pages 166–175, August 1994.

[125] G. D. Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Clarendon Press, 1985.

[126] Julius O. Smith. Physical modeling synthesis update. *Computer Music Journal*, 20(2):44–56, Summer 1996.

[127] Neil Spring and Rich Wolski. Application level scheduling of gene sequence comparison on meta-computers. In *Proceedings of the 12th ACM International Conference on Supercomputing (ICS '98)*, July 1998.

[128] J. Stankovic and K. Ramamritham. *Hard Real-Time Systems*. IEEE Computer Society Press, 1988.

[129] Sun Microsystems, Inc. Java remote method invocation specification, 1997. Available via http://java.sun.com.

[130] Murad S. Taqqu, Vadim Teverovsky, and Walter Willinger. Estimators for long-range dependence: An empirical study. *Fractals*, 3(4):785–798, 1995.

[131] The Open Group. *DCE 1.2.2: Introduction to OSF DCE*. The Open Group, September 1997. http://www.opengroup.org/pubs/catalog/f201.htm.

[132] Howell Tong. *Threshold Models in Non-linear Time Series Analysis*. Number 21 in Lecture Notes in Statistics. Springer-Verlag, 1983.

[133] Jeffrey S. Vetter and Daniel A. Reed. Managing performance analysis with dynamic statistical projection pursuit. In *Proceedings of Supercomputing '99 (SC '99)*, November 1999.

[134] Steve Vinoski. Distributed object computing with CORBA. *C++ Report*, July/August 1993.

[135] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. 19th Intl. Conf.. on Computer Architecture*, May 1992.

[136] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*. Usenix, 1994.

[137] Walter Willinger, Murad S. Taqqu, Will E. Leland, and Daniel V. Wilson. Self-similarity in high-speed packet traffic: Analysis and modeling of ethernet traffic measurements. *Statistical Science*, 10(1):67–85, January 1995.

[138] Walter Willinger, Murad S. Taqqu, Robert Sherman, and Daniel V. Wilson. Self-similarity through high-variability: Statistical analysis of ethernet lan traffic at the source level. In *Proceedings of ACM SIGCOMM '95*, pages 100–113, 1995.

[139] Victor Fay Wolfe, John K. Black, Bhavani Thuraisingham, and Peter Krupp. Real-time method invocations in distributed environments. Technical Report 95-244, Department of Computer Science, University of Rhode Island, January 1996.

[140] Rich Wolski. Forecasting network performance to support dynamic scheduling using the network weather service. In *Proceedings of the 6th High-Performance Distributed Computing Conference (HPDC97)*, pages 316–325, August 1997. extended version available as UCSD Technical Report TR-CS96-494.

[141] Rich Wolski, Neil Spring, and Jim Hayes. Predicting the CPU availability of time-shared unix systems. In *Proceedings of the Eighth IEEE Symposium on High Performance Distributed Computing HPDC99*, pages 105–112. IEEE, August 1999. Earlier version available as UCSD Technical Report Number CS98-602.

[142] Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service: A distributed resource performance forecasting system. *Journal of Future Generation Computing Systems*, 1999. To appear. A version is also available as UC-San Diego technical report number TR-CS98-599.

[143] Gregory Wornell. *Signal Processing with Fractals: A Wavelet-Based Approach*. Prentice Hall, 1995.

[144] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. On-line extraction of SCSI disk drive parameters. In *Proceedings of ACM SIGMETRICS '95*, pages 146–156, May 1995.

[145] John A. Zinky, David E. Bakken, and Richard E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1):55–73, April 1997.